NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DETERMINING THE LOCATION OF AN
OBSERVER WITH RESPECT TO
AERIAL PHOTOGRAPHS

by

Jill Donahue Wolfe

December 1988

Thesis Advisor: Neil C. Rowe

Approved for public release; distribution is unlimited.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) Code 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000
8a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)

DETERMINING THE LOCATION OF AN OBSERVER WITH RESPECT TO AERIAL PHOTOGRAPHS

12. PERSONAL AUTHOR(S) Wolfe, Jill D.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1988 December	81

16 SUPPLEMENTARY NOTATION

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Computer vision; motion of an observer; aerial photo interpretation.

19 ABSTRACT (*Continue on reverse if necessary and identify by block number*)

In this study, the possibility of using a computer to detect the motion of an observer by comparing two successive aerial photographs is examined. The purpose of the study was to experiment with a technique for finding a point common to both images. The technique presented uses only sharp boundary lines and their distribution in the images to produce the "primal sketches" of the image. Once the "primal sketches" are made, the original images are not referred to again. A point common to both images is identified by comparing the number of cells with strong gradient magnitudes and their distribution in 3-pixel by 3-pixel blocks. The technique produces excellent results in analyzing simulated successive photographs, suggesting good results with photographs that are taken in succession by a moving observer. Compared with other work on image correlation and object identification, the technique uses fewer features (only two) in its primal sketches, and it does not need any human intervention. Possible applications are photo interpretation, high-altitude navigation, and underwater station-keeping.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT [X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT [ ] DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Neil C. Rowe	22b. TELEPHONE (*Include Area Code*) 408-646-2462 22c. OFFICE SYMBOL Code 52RP

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited

Determining the Location of an Observer With Respect to Aerial Photographs

by

Jill Donahue Wolfe
Lieutenant Commander, United States Navy
B.A., East Tennessee State University, 1975

Submitted in partital fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

ABSTRACT

In this study, the possibility of using a computer to detect the motion of an observer by comparing two successive aerial photographs is examined. The purpose of the study was to experiment with a technique for finding a point common to both images. The technique presented uses only sharp boundary lines and their distribution in the images to produce the "primal sketches" of the image. Once the "primal sketches" are made, the original images are not referred to again. A point common to both images is identified by comparing the number of cells with strong gradient magnitudes and their distribution in 3-pixel by 3-pixel blocks. The technique produces excellent results in analyzing simulated successive photographs, suggesting good results with photographs that are taken in succession by a moving observer. Compared with other work on image correlation and object identification, the technique uses fewer features (only two) in its primal sketches, and it does not need any human intervention. Possible applications are photo interpretation, high-altitude navigation, and underwater station-keeping.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I wish to acknowledge the help, guidance, and support provided by my father, to whose memory this thesis is dedicated.

I. INTRODUCTION

This thesis addresses the problem of detecting motion of an observer above the earth's surface. Specifically, it explores ways to detect the motion of an observer by comparing two successive images taken by the observer, identifying a feature present in both images, and then measuring the distance that the feature has moved from the first image to the second. The images are two black and white photographs converted to digitized images, the basic element of which is a picture element (pixel). As long as the observer has not rotated between taking two images, comparing the digitized images pixel by pixel is easily done, but this brute force method of comparing images is not efficient unless a match is found on the first scan of each image (best case). In the worst case, for example, it would take 800 scans per image and 1,280,000 comparisons to compare a twenty-pixel by twenty-pixel image to a forty-pixel by forty-pixel image. Therefore, optimization of the comparison is a major part of the problem. The idea is to compare only those parts of the images that are statistically interesting instead of comparing every pixel in one image to every pixel in the other.

The military has long been interested in automated analysis of reconnaissance photographs. Having a computer interpret an image eliminates the need to train a human to do it. Computers can include image enhancement as part of the analysis, bringing out details impossible to detect with the human eye. Computers can also detect very small changes in successive images; such changes must be significantly greater before the human eye can see it. High altitude navigation can be aided by a computer that can detect the motion of the aircraft over the ground by comparing two successive photographs taken by the aircraft. Such a computer could accurately and quickly compute speed over the ground.

Computers have been used successfully to solve various visual problems. A computer can make the precise measurements needed for detecting motion. In addition, a computer can easily and quickly do any necessary preprocessing of digitized images and can quickly compare the statistical information gathered by the preprocessing. In particular, a computer optimized to work with the language LISP can carry out both mathematical computations and list processing very rapidly; array processing is also fast in LISP. Fast mathematical calculations and fast list and array processing are both needed for detecting motion with the techniques presented in this thesis. A LISP computer could fit aboard some types of aircraft, and any such ground-based computer could be used to

analyze photographs after the aircraft's mission is completed. For these reasons, solving this problem on a computer in LISP became the basis for this thesis.

New ideas used in solving the problem include:

- Eliciting two sets of special statistics from each image after first finding the gradient magnitude for each pixel. The gradient used is the square of the sum of the differences between a pixel and its right and bottom neighbors; an edge exists at that pixel if the gradient magnitude of the pixel exceeds a certain amount (called the threshold). The sets of statistics elicited are the number of edges in each of the three-pixel by three-pixel blocks into which the images are divided, and a unique code value assigned each block based on the number and position of the edges contained in it. The blocks into which the images are divided are not adjacent three-pixel by three-pixel blocks but rather overlapping blocks--a forty by forty image would contain 1369 such blocks; each pixel in the images, with the exception of the pixels in the last two columns and rows, is the upper left-hand corner of one of these blocks.

- Searching for a match in the two images by finding two points (one from each image) that exhibit the exact same values for three different statistics about one feature, the edges in the two logical three-pixel by three-pixel blocks defined by the two points: the number of edges in the three-pixel by three-pixel blocks, the values assigned to the blocks based on the positions of the edges within them, and the direction and distance of these blocks from all other blocks, in their respective images, that contain the same number of edges.

The problem is solved simply, using the single feature, a few straightforward mathematical calculations, and some list processing. The program also automatically chooses the threshold. In contrast, solutions found by others to similar problems require multiple features and human interaction with the computer to teach it and to choose the features and thresholds to be used.

The remaining chapters of this thesis explain the program. Chapter II is a discussion of related theses that served as points of departure for the program and of papers that provided insights into techniques used to solve other visual problems. Chapter III discusses the various assumptions made in approaching the problem, expands on the situation that the program models, and explains how motion of the observer is represented to the program. Chapter IV describes the program in some detail. Chapter V discusses the results produced by the program. Chapter VI provides the conclusions drawn from the results.

II. SURVEY OF PREVIOUS WORK

A. INTRODUCTION

In sections B and C, some ideas that have been used to solve various vision problems are presented. In sections D and E, two theses that served as points of departure for this thesis are briefly explained.

B. GENERAL APPROACHES TO COMPUTER VISION

A convenient way to think of an image is as a function giving the gray level at every point on the image plane. Gray levels vary from 0 (totally black) to 1 (maximum brightness). Since an image surface is two-dimensional, a Cartesian coordinate system can be used to assign x and y coordinates to the image surface. The image can be divided into square cells, each of which is assigned x-y coordinates. Each square cell is a pixel (short for "picture element"), the image's smallest unit of measure. Further, the image can be reproduced on a graphics screen by displaying the average gray level for each pixel of the image [Ref. 1:p. 89]. These average gray levels can also be stored in arrays with indices that correspond directly to the x and y coordinates of the image.

Vision processing can be divided into two phases: early and late. In early vision, a primal sketch is produced. A primal sketch is a database of data structures, each of which describes a feature. Primal sketches contain three kinds of features, the most interesting of which is an edge, for purposes of this thesis. An edge is a small patch where the gray level goes from dark to light [Ref. 1:pp. 99-100]. The existence of an edge depends on the degree to which the gray level changes from dark (0 at the blackest) to light (1 at the whitest); that is, a threshold may be defined that allows the difference between the gray levels of two pixels to be called an edge if the difference is only .001, while in other cases, the difference may have to be as much as .4 for it to be designated an edge.

It may be that, once a primal sketch is produced, it is not necessary to look at the original image again [Ref. 1:p. 99]. That is the approach tried in this thesis, as it was in LT Jean Sando's work [Ref. 2]. Reference 3 describes an approach to stereo image matching which also uses only the primal sketches produced. The major disadvantage of the method described in [Ref. 2] and that described in [Ref. 3] is that human interaction is required to teach the computer what to extract from the image.

C. STEREO MATCHING AND MOTION DETECTION

Stereo matching involves finding corresponding points in two slightly different images of the same scene. Since the two images are taken from different positions, some points visible in one image may be obstructed in the other. According to Roman, Laine, and Cox [Ref. 3],

> Most approaches [to stereo vision] may be classified as area-based or feature-based. Area-based techniques rely on the surface continuity assumption and often involve correlation- based matching. Feature-based approaches focus on intensity variations that correspond to physical and geometric properties and intensity anomalies which may not have any physical relevance. Matching is often doneat the symbolic level. Much effort has been devoted to the study of feature-based techniques because they provide better localization and exploit more contextual information. [Ref. 3:p. 171]

They go on to offer an incremental matching strategy, in which the user selects features which he thinks stand the best chance of being matched successfully. Among the features that the user can choose are edge strength, orientation, length, and texture; the feature chosen can be used alone to find the match or in combination with other features [Ref 3:pp. 171-172].

Stereo matching is very similar to the problem addressed in this thesis. Both stereo matching and the program presented in this thesis try to find matching edges in two similar images. This program is very successful in finding matches using one feature only, and it does not require

interaction with the user. Other research shows that the same technique used to find matching edges in two similar images also can detect the motion of the observer that took the images.

D. DETECTING ROADS IN AERIAL PHOTOGRAPHS

The purpose of LT Jean Sando's thesis [Ref. 2] was to find roads in an aerial photograph by using texture to identify the regions of a digitized image of the photograph. The program gathered statistical information that provided evidence as to which textures indicated road areas and which textures signalled off-road areas. The images had to be divided into units, called windows, because texture has meaning only in relation to groups of pixels or points. In general, the size and shape of windows depends on the size and shape of the objects to be identified [Ref. 2:p. 12]. Here, two categories of windows were designated. The inner window was the area being classified; the outer window was centered on the inner window but was larger. To quote LT Sando,

> The three features chosen for this study were the mean of the inner window, the variance of the inner window and the variance of the outer window. The mean of the outer window was not used because early tests indicated the results would be virtually identical to the result from the inner window mean. The mean of the inner window defines the gray level of the window. The gray level is important because in most cases there is a distinct difference in this feature between the two textures. The variance from the inner window defines the granularity, or roughness of the window. The outer window is used to determine if the granularity varies as the size of the window approaches or exceeds the size of the object, in

> this case the road. These moments were chosen because the calculations were relatively simple and they represent features that intuitively suggest the different properties the human eye might pick out. [Ref. 2:p. 16]

Each window was identified by using two different methods. In the first, each scan window was tested against each of the three features separately; if any of the features indicated the scan window to be a road area, it was deemed to be a road. In the second method, called a Gaussian scan, all three features were used together to decide whether the window was a road area or an off-road area; in essence, the value of each feature in the window added to the evidence until there was enough evidence to determine whether a road or off-road area was defined by the window. LT Sando concluded that the Gaussian scan produced the better results. [Ref. 2:pp. 26-27]

Using LT Sando's proposition that building up the evidence in favor of a conclusion produced more reliable results, the program presented in this thesis first finds strong evidence that two blocks (one from each image) are the same, based on their having the same number of edges within them. Then it attempts to confirm the hypothesis by comparing where the edges lie within the blocks; if all the edges in one block are in the same positions as all the edges in the other block, the evidence is strong enough to claim that they are identical and represent the same point on the earth's surface.

E. STATION KEEPING USING BOTTOM-TRACKING SONAR

LT Chet Hartley's thesis [Ref. 4] was an attempt to solve the problem of detecting the motion of an Autonomous Underwater Vehicle as it tried to maintain its position over a particular spot of the ocean floor, a process known as station-keeping. LT Hartley used acoustic signal data instead of visual. He stored the results of bottom scans (depth information) in arrays. Successive arrays were compared by initially laying one on top of the other, center to center. Next, the center three-cell by three-cell block (mask) in the top image was compared to the center three-cell by three-cell block (mask) in the bottom image by finding the difference between the value of one of the nine cells in the top image mask and the value of the corresponding cell in the bottom image mask and then squaring the difference. Specifically,

$$(D1(xi,yi) - (D2(xi,yi))^2 \qquad (2.1)$$

where D1 is the top image and D2 is the bottom image, and i refers to the corresponding cells in each of the three-cell by three-cell masks. This was done for each pair of corresponding cells in the two masks. He proposed that if the formula produced the smallest square when applied to the center cells of the two masks, those cells represented the same spot on the ocean floor. Otherwise, the masks were shifted and the comparisons repeated until the result of

applying the formula to the center cells of the masks was smaller than the result of applying the formula to any other corresponding cells. These two cells were considered to represent the same spot on the ocean bottom; the distance and the direction of the bottom mask's center cell from the bottom array's center was the distance and direction that the Autonomous Underwater Vehicle had moved between scans. [Ref. 4:pp. 44-49]

LT Hartley did not need to collect statistics about both arrays. Instead he compared actual cells (depth data), the equivalent of comparing gray levels to find a match between two digitized images. He also started the comparisons at the center of both arrays, a logical idea since his simulated Autonomous Underwater Vehicle was centered over the spot represented by the center array cells when the scans were made. He never had to consider any but the center cells of the top array, but in the worst case he might have to look at all the cells in the bottom array. All in all, his technique was efficient, each set of comparisons consisting only of squaring the differences between nine pairs of values.

III. DESCRIPTION OF APPLICATION

A. INTRODUCTION

Interpretation of aerial photographs came of age during World War II, during which 80% of all military intelligence was derived from aerial photographs [Ref. 5:p. 217]. Even though more than forty years have passed, aerial-photograph interpretation remains largely a manual task [Ref. 5:p. 219], involving

> ...careful comparison of photo coverage over days, weeks, and even months; use of stereo vision; measurement of images to the tenth part of a millimeter with a high-power magnifier fitted with a graticule; and above all, visual imagination. [Ref. 6:p. 63]

To find the same point in two nearly identical photographs taken at the same height involves placing one photo under one eyepiece and the other under the second eyepiece and moving the photographs until they produce a clear, stereo image. Manual measurements can then be taken to determine how far the camera moved between photographs. This thesis explores a means of automating these procedures.

B. APPLICATION

The program presented in this thesis compares two aerial photographs taken only moments apart so that both contain a

portion of the same scene; then it tries to detect exactly where that portion appears in each in photograph and how far that portion has shifted from its position in the first photograph to its position in the second.

C. ASSUMPTIONS

The following assumptions were made:

- The pictures analyzed have sufficient diversity in gray levels for edges to appear; i.e., there should be several places in the pictures where the square of the difference between gray levels of adjacent pixels is significant.
- The motion of the observer is in one plane only, a plane parallel to the plane of the camera.
- The light source provides the same amount of illumination for two succeeding photographs, making it possible to use the same threshold for both images. The threshold is the quantity, between 0 and 1, with which the presence of edges is detected; i.e., when the square of the difference between the gray levels of adjacent pixels equals or exceeds that quantity, an edge is present.
- The relatively small images enable sufficient enough testing of the techniques to provide confidence that the techniques could be used on larger images.
- No highly regular, repeated patterns appear in the photographs; e.g., parking lots containing many cars.
- No rotation of the picture frame occurs between the two pictures.

D. SITUATION MODELED

The program presented in this thesis models the following scenario: a photographic system aboard an aircraft takes two successive photographs at a known height and a known amount of time apart but close enough in time that both photographs

contain the same portion of ground. Digitized representations of the two images are processed and compared, and the distance traveled by the aircraft is determined by how far the portion common to both images has shifted from its position in the first image to its position in the second.

E. IMAGES USED

Figures III-1 and III-2 show the two 500 by 500 images that were manipulated to produce the smaller (40 by 40) images (also shown) that were used to test the program. The reduced images covered the same areas as the 500 by 500 images, but in less detail.

Obviously, Image 1b cannot be compared to Image 2b since they have no common ground. But, for testing purposes, each image can be partitioned into a pair of images that can be compared. The program used this idea to produce two images to be compared from a single image.

Image 1a

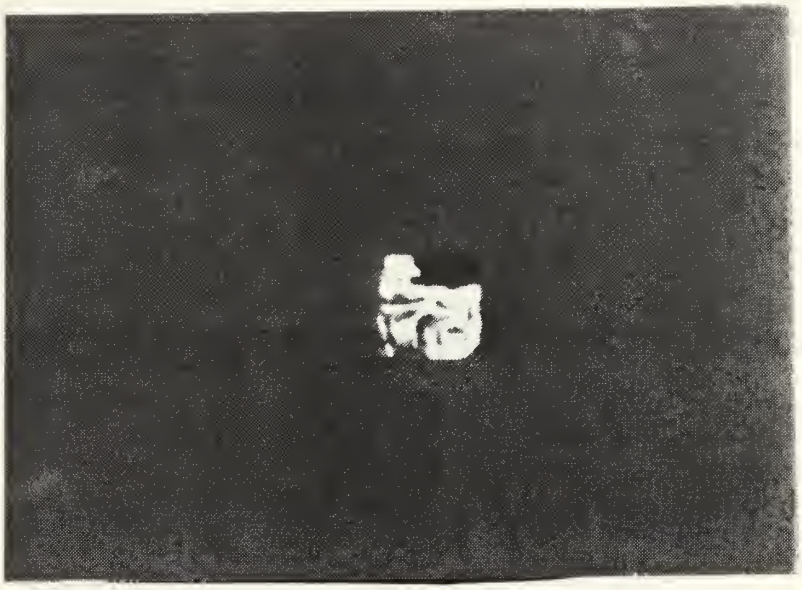

Image 1b

Figure III-1. First Image Used to Test Program.

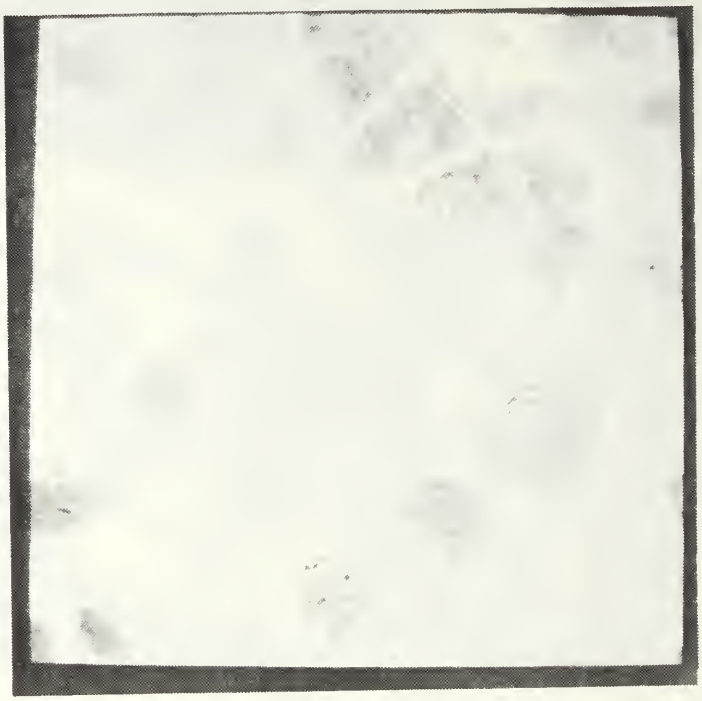

Image 2a

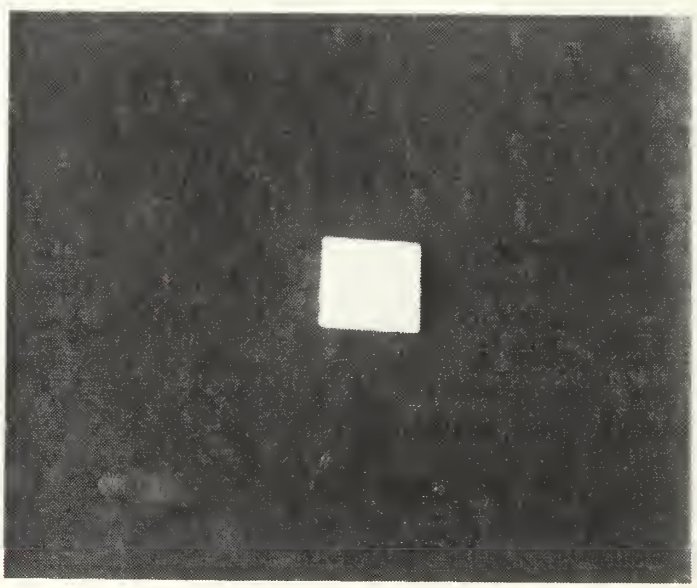

Image 2b

Figure III-2. Second Image Used to Test Program.

IV. IMPLEMENTATION

A. INTRODUCTION

The program is written in Common LISP, in the Symbolics dialect. It runs on a Symbolics LISP machine; in particular, it was run on a Symbolics 3675 with five megabytes of memory, a Symbolics 3650 with five megabytes of memory, and a Symbolics 3640 with 1024 kilobytes of memory.

Like LT Hartley's program, this program tries to detect the motion of an observer over some surface. Also like LT Hartley's, it focuses attention on overlapping three-cell by three-cell blocks. It uses a formula similar to formula 2.1 to find the gradient magnitude of each pixel in the images. Like LT Sando's program, it tries to solve the problem visually, as opposed to acoustically. Also like LT Sando's, it creates the primal sketches by doing statistical analysis over both images entirely and by starting its scans in the upper left-hand corner of the images. However, the features in the primal sketches were different from those in LT Sando's.

B. PROGRAM INPUT

There are three inputs to the program:

- The name of the file that holds the forty-pixel by forty-pixel image that represents the "first" photograph taken by the observer.
- The name of a file if the user wants the screen output stored to a file.
- The user-chosen x-y coordinate that defines the upper left-hand corner of the subimage that represents the "second" photograph taken by the observer.

Note that the "second image is actually implemented as a part of the "first" image for test purposes but will not be in a real application. From here forward, the "first" image will be referred to as the original image (**orig-image** in the program itself) and the "second" image as the subimage.

Instead of underwater images as in LT Hartley's thesis, this thesis used aerial photographs of different parts of Fort Hunter-Liggett, California. Each photograph was digitized into a 500 by 500 image and stored in an image file. Because processing a 500 by 500 image could be unnecessarily time-consuming for test purposes, the files were further reduced to forty by forty image files by keeping every twelfth column and row of the digitized images, resulting in images that are lower-resolution, miniature versions of the original 500 by 500 images. Each reduced image was placed in its own file that could be used by the program. A single file was used to produce the two images compared by the program. The

second image was some twenty-pixel-by-twenty-pixel section of the file while the first image was the entire file. The program processes the first and second images, extracting statistical information about all possible three-by-three blocks in the images. The program uses the statistics to find pairs of statistically similar blocks in both pictures. The difference between the indices into the statistical arrays for these blocks yields the offset of the upper left-hand corner of the second image from the upper left-hand corner of the first image.

C. PROGRAM OUTPUT

The output of the program is the registration of the upper left-hand corner of the subimage within the original image.

D. MAJOR INTERACTIONS

Figure IV-1 is a block diagram that shows the main interactions within my program.

The major high-level function--process_images--first calls all the preprocessing functions--those that do the statistical analyses of the images and store the information in various arrays. Function **calculate_gradient** produces a gradient array for the original image and one for the subimage. Function **select_threshold** uses these gradient arrays to select the threshold to be used on both images. Using the threshold,

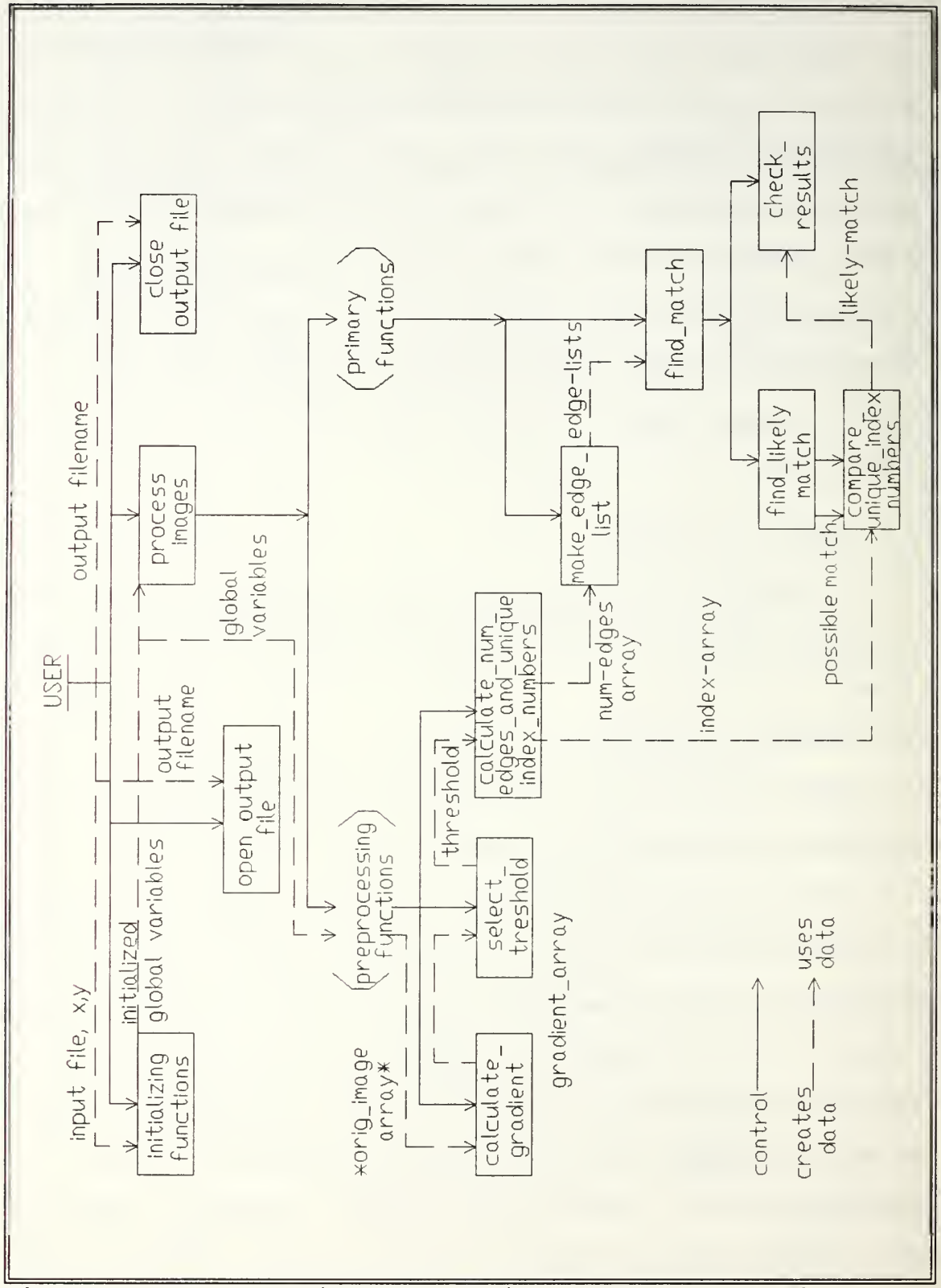

Figure IV-1. Block Diagram of Main Program Interactions.

calculate_num_edges_and_unique_index_numbers produces a **num-edges-array** and **unique-index-number** array for each image.

Next, **process_images** calls the primary functions, those that do the comparisons and list processing that result in finding the registration of the subimage within the original image. Function **make-edge-list** puts into list form certain indices into certain arrays. Function **find_match** sends these lists to **find_likely_match.** If **find_likely_match** finds a possible match in these lists, it calls **compare_unique_index_numbers**, which looks for even stronger evidence that the possible match is a likely match. If that stronger evidence is found, **find_match** calls **check_results**; if it is not, **find_match** prints a message that says that there were no matches in the edge lists.

E. FUNCTIONS WRITTEN BY OTHERS

All the functions used to display the images on the graphics screen were written by CPT Jim Zanoli, U. S. Army.

- **make_color_window** produces a window labeled "IMAGE ANALYSIS" on the graphics screen when the program is loaded.
- **make_blue_window** makes blue the window produced by **make_color_window.**
- **color_pixel** is an I/O function that sends the Red-Green-Blue (RGB) values to a specific pixel on the graphics screen. For the purposes of this thesis, the RGB values of each pixel are the same, creating gray levels that match the gray levels in the black and white digitized image.

- **display_image** can be used to display the image once it is stored in an array.

F. DATA STRUCTURES

Lists and arrays are used in this thesis. LISP is particularly adept at handling and manipulating lists; they are LISP's natural data structure. Arrays are the logical structure for handling image data.

There are seven major arrays used in the program:

- ***orig-image-array*** is an n by n array that holds the gray level values of the original image. To save memory, part of it also acts as the virtual array that holds the gray levels of the subimage. When the program needs to access the subimage in order to create the ***subimage-gradient-array***, it is actually accessing that part of the ***orig-image-array*** that is the subimage. This is the only time that the program uses the otherwise privileged information as to where the subimage lies within the original image; the artificiality of using the ***orig-image-array*** as the subimage's virtual image array makes this disclosure necessary. It enters the ***orig-image-array*** at the upper left-hand corner of the subimage (the coordinates of that point are an input to the program) and processes only those pixels that make up the subimage.

- ***orig-image-gradient-array*** is an n-1 by n-1 array that stores the results of finding the gradient magnitude of each pixel in the original image.

- ***orig-image-num-edges-array*** is an n-3 by n-3 array that holds the number of edges found in each overlapping three-cell by three-cell block into which the ***orig-image-gradient-array*** is virtually divided. Every ***orig-image-gradient-array*** cell that has at least two neighbors to its right and at least two neighbors to its left is the upper left-hand corner of a three-cell by three-cell block; therefore, in an n by n image, there are ((n-3) * (n-3)) such blocks. The blocks into which the ***orig-image-gradient-array*** and its corresponding ***subimage-gradient-array*** are divided are the units about which statistics are gathered and comparisons are made.

- ***orig-image-index-array*** is an n-3 by n-3 array that stores the unique numbers representing the three-cell by three-cell blocks into which the ***orig-image-gradient-array*** is virtually divided. The formula used to compute these unique numbers is described in the section entitled PROGRAM COMPONENTS.

- ***subimage-gradient-array*** is a k-1 by k-1 array that holds the results of finding the gradient of each pixel in the subimage, like the array ***orig-image-gradient-array*** above. **k** is the length of one side of the subimage.

- ***subimage-num-edges-array*** is a k-3 by k-3 array for the subimage, like the array ***orig-image-num-edges-array*** above.

- ***subimage-index-array*** is a k-3 by k-3 array for the subimage, like the array ***orig-image-index-array*** above.

The program uses four main lists:

- ***orig-image-five-edge-list*** is a list of the subscripts into ***orig-image-num-edges-array*** whose cells indexed by the subscripts yield the number five. These cells represent the three-cell by three-cell blocks of the ***orig-image-gradient-array*** that have five edges after the threshold is applied.

- ***orig-image-six-edge-list*** is similar to the list ***orig-image-five-edge-list***, except for six-edge cells.

- ***subimage-five-edge-list*** is like the list ***orig-image-five-edge-list***, except for the subimage.

- ***subimage-six-edges-list*** is like the list ***orig-image-six-edge-list***, except for the subimage.

G. PROGRAM COMPONENTS

The program is divided into the major sections shown in the block diagram of Figure IV-1. In addition, there are several miscellaneous functions that serve to support the user

in better understanding what the program is doing. These functions are found at the very end of the program.

1. Initializing Functions

Five small functions are in this section. Three, described below, must be called by the user--**make_image_array**, **initialize**, and **set_up**. The function **set_up** calls **make_sub_image**, and **make_image_array** calls the function read_file.

- **make_image_array** reads the image file with the help of the function **read_file** and stores the gray level values in ***orig-image-array***.
- **initialize** initializes the various global variables used in the program.
- **set_up** calls **make_sub_image** to assign values to ***subimage-x*** and ***subimage-y*** and to make sure that a subimage that starts at the point defined by ***subimage-x*** and ***subimage-y*** fits entirely within the original image. This error-checking facility ensures that the virtual subimage is completely inside the original image so that the array subscripts of the virtual subimage will never be out of bounds of the ***orig-image-array***.

2. Preprocessing Functions

These functions carry out the statistical analysis of the original image and of the subimage. This statistical analysis enables the program to find statistically interesting parts of the images to process and compare. Comparing the two images pixel by pixel is thereby avoided; the program only compares small sections of the images instead. The arrays created by these functions are then processed and compared by the primary functions, while the original image and the

subimage are never looked at again. In fact, the only function that actually scans the images is **calculate_gradient**.

Figures IV-2 through IV-5 are the arrays that hold the statistical information about the subimage taken from Image 1 when ***subimage-x*** is 5 and ***subimage-y*** is 10 (i.e., the subimage's upper left-hand corner is at the intersection of the sixth column and eleventh row of the original image when the first row and first column are numbered zero as is the convention in computer science). The threshold above which the difference squared amongst three neighboring pixels produces an edge is .35. The functions that produce the arrays and select the threshold are:

- **calculate_gradient (size x y gradient-array)**--scanning the ***orig-image-array***, calculates the gradient magnitude for each cell in the ***orig-image-array*** starting at the array cell indexed by x and y. The formula used is the sum of the square of the differences between the gray level of a pixel and two of its neighbors:
  $$\text{sqrt}[(Pi,j - Pi+1,j)^2 + (Pi,j - Pi,j-1)^2] \quad (4.1)$$
  where i and j take on the values of zero to the number of pixels minus one in a row or column (n-1 or k-1, as appropriate). Size is the length of a side of the original image (n) or the length of a side of the subimage (k). x and y are both zero if the image being processed is the original image; x is ***subimage-x*** and y is ***subimage-y*** if the image being processed is the subimage. See Figure IV-2.

- **select_threshold**--this function selects the threshold that produces as near to m five-edge blocks in the subimage as possible, where m is 2.5% of the total number of pixels in the subimage. It tries increments of .1 until it finds two successive thresholds, one of which produces less than m five-edge blocks and the other more than m five-edge blocks. Minor interpolation is done to fine-tune the threshold as much as possible. The resulting threshold is used to preprocess both the subimage and the original image. This function calls

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
.02	.05	.22	.52	.12	.00	.00	.01	.00	.01	.01	.00	.01	.01	.01	.02	.01	.01
.03	.04	.31	.35	.33	.00	.01	.01	.00	.00	.00	.01	.01	.01	.00	.01	.01	.01
.07	.03	.30	.05	.54	.21	.02	.00	.00	.01	.01	.00	.00	.01	.01	.01	.02	.01
.05	.08	.33	.25	.53	.60	.52	.01	.01	.01	.01	.00	.01	.01	.01	.00	.01	.00
.11	.16	.26	.27	.09	.25	.54	.02	.01	.00	.00	.00	.01	.00	.01	.01	.01	.00
.23	.18	.32	.38	.62	.71	.93	.03	.01	.00	.00	.00	.00	.00	.00	.01	.01	.01
.43	.49	.67	.09	.12	.04	.07	.00	.00	.00	.00	.00	.01	.00	.01	.00	.00	.01
.16	.65	.24	.20	.01	.00	.01	.00	.00	.00	.00	.01	.01	.01	.00	.00	.01	.00
.45	.26	.36	.08	.00	.01	.01	.01	.01	.00	.00	.00	.01	.01	.00	.01	.00	.00
.38	.24	.39	.20	.06	.04	.01	.01	.01	.01	.01	.02	.03	.02	.02	.01	.01	.01
.59	.30	.35	.57	.73	.69	.73	.76	.67	.51	.43	.65	.77	.74	.69	.71	.63	.40
.25	.11	.08	.04	.12	.18	.13	.13	.23	.19	.29	.24	.23	.06	.10	.11	.24	.36
.32	.22	.01	.06	.02	.12	.19	.17	.17	.52	.28	.14	.20	.14	.16	.11	.20	.24
.40	.45	.53	.30	.25	.35	.33	.43	.53	.47	.61	.16	.20	.26	.18	.26	.39	.06
.08	.08	.30	.32	.33	.29	.17	.19	.38	.38	.30	.12	.11	.08	.44	.31	.44	.23
.06	.14	.11	.12	.28	.14	.19	.09	.04	.60	.04	.14	.07	.73	.34	.01	.21	.13
.14	.08	.11	.14	.11	.04	.31	.42	.52	.27	.12	.04	.61	.04	.27	.18	.05	.15
.21	.34	.39	.40	.52	.61	.24	.12	.22	.11	.27	.26	.44	.27	.21	.18	.26	.35
.31	.15	.08	.16	.10	.37	.39	.08	.09	.50	.32	.13	.12	.18	.26	.26	.29	.15

Figure IV-2. *subimage-gradient-array* for Image 1.

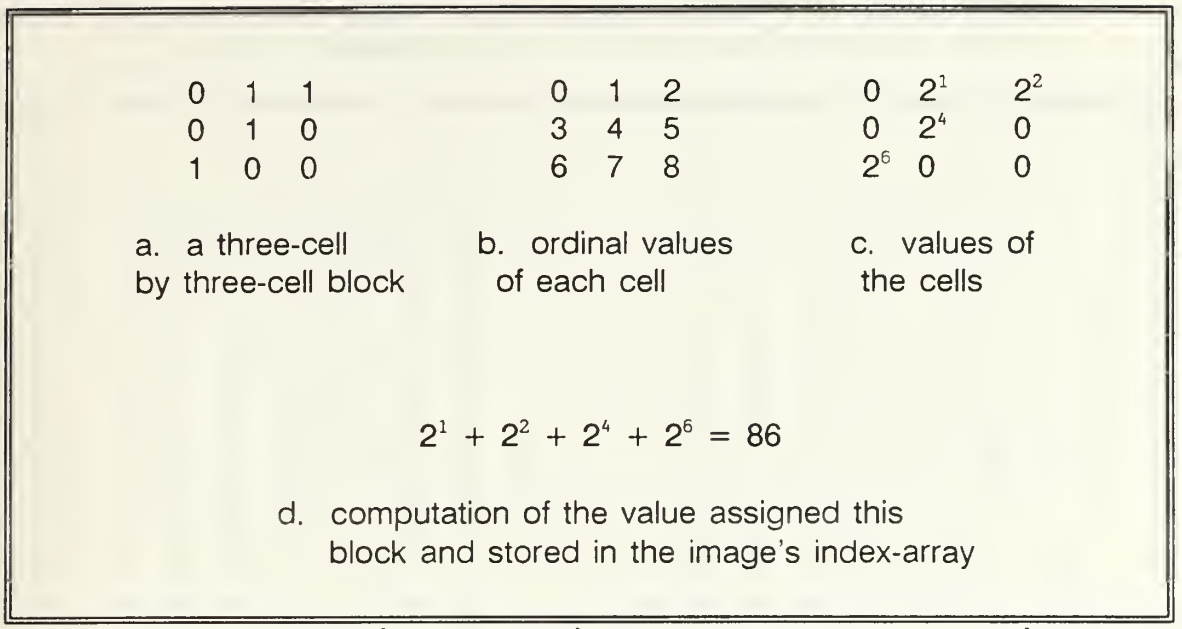

Figure IV-3. Computing the Unique Index Value of a Nine-Cell Block.

functions **make_edge_histogram** and **calculate_num_edges_and_unique_index_numbers.**

- **calculate_num_edges_and_unique_index_numbers (gradient-array size threshold unique-index-num-array num-edges-array)**--this function produces two arrays for each image. For the original image, it produces ***orig-image-index-array*** and ***orig-image-num-edges-array***. For the subimage, it produces ***subimage-index-array*** and ***subimage-num-edges-array***. Starting at the upper left-hand corner of the appropriate gradient array, the function logically divides the gradient array into overlapping three-cell by three-cell block. Each cell in the two arrays produced by this function is the upper left-hand corner of some three-cell by three-cell block in one of the gradient arrays. Those cells in the block whose values exceed those of the threshold are flagged as edges. The number of edges in each three-cell by three-cell block is stored in the appropriate **num-edges-array**, indexed by the same indices as those that define the gradient array cell in the upper left-hand corner of the block. See Figure IV-4. Simultaneously, each edge cell in each three-cell by three-cell block is assigned a value of 2 raised to some power; then the values of each cell in the nine-cell block are added together, giving the block a unique value

(Figure IV-3). The values for these blocks are stored in the appropriate index array (Figure IV-5).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	3	3	4	3	4	3	3	3	3	3	3	3	3	3	3
1	0	1	2	2	2	2	3	3	3	3	3	4	4	4	3	3	3
2	0	0	1	2	3	2	1	0	0	1	2	4	4	3	1	0	0
3	0	0	1	2	3	2	1	0	0	1	3	6	6	4	1	0	1
4	0	0	1	2	3	2	1	0	0	1	3	6	6	4	1	1	2
5	0	0	0	0	0	0	0	0	0	1	3	5	4	2	1	2	3
6	0	0	0	0	1	2	3	3	2	2	2	3	2	1	2	3	3
7	3	3	2	1	1	2	3	4	4	4	3	3	2	1	2	2	2
8	3	3	2	1	1	2	3	4	4	3	2	2	2	1	1	1	1
9	3	3	2	1	0	0	0	1	2	3	4	4	3	1	0	0	0
10	0	0	0	0	0	0	0	0	1	3	5	4	2	0	0	0	0
11	0	0	0	0	0	0	0	1	3	5	5	3	1	0	0	0	0
12	0	0	0	0	0	1	2	4	5	5	3	1	0	0	0	0	0
13	0	0	0	0	1	2	4	5	5	3	1	0	0	0	0	0	0
14	1	2	3	2	2	2	4	4	3	1	0	0	0	0	0	0	0
15	2	3	3	2	2	1	3	2	2	0	0	0	0	0	0	0	0
16	2	3	3	2	1	0	2	2	2	0	0	0	0	0	0	0	0

Figure IV-4. *subimage-num-edges-array* for Image 1.

- **make_histogram (num-edges-array size)**--function **make_histogram** is called by function **select_threshold** to find out how many five-edge blocks there are in the gradient array of the image in question. The output is a list whose values represent the total number of 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 edges found in the three-cell by three-cell blocks into which the gradient array of the image in question is divided. For example, the output might look like this for the original image:

  (475 215 208 230 149 66 20 6 0 0)

  In this case, the original image has 475 blocks with zero edges in them, 208 with two edges, and 66 blocks with five edges. Computing the total number of blocks with particular numbers of edges in them indicates how many blocks hold the number of edges of statistical interest. The threshold dictates how many edges appear in a particular block; by knowing the total number of five-edge blocks that a threshold produces, the threshold can be adjusted to produce the desired number of five-edge blocks.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	2	33	52	26	45	50	57	56	56	56	56	56	56	56	56	56	56
1	0	4	6	3	5	6	7	7	7	7	7	263	135	71	7	7	7
2	0	0	256	384	448	192	64	0	0	256	384	480	464	200	64	0	0
3	0	0	32	48	56	24	8	0	0	32	304	444	250	89	8	0	256
4	0	0	4	6	7	3	1	0	0	4	38	311	159	75	1	256	160
5	0	0	0	0	0	0	0	0	0	256	388	230	83	9	256	160	84
6	0	0	0	0	256	384	448	448	192	96	48	28	10	1	288	148	74
7	448	448	192	64	32	48	56	312	408	204	70	259	129	64	36	18	9
8	56	56	24	8	4	6	7	39	51	25	264	160	80	8	4	2	1
9	7	7	3	1	0	0	0	4	6	259	417	212	74	1	0	0	0
10	0	0	0	0	0	0	0	0	256	416	244	90	9	0	0	0	0
11	0	0	0	0	0	0	0	256	416	244	94	11	1	0	0	0	0
12	0	0	0	0	0	256	384	480	244	94	11	1	0	0	0	0	0
13	0	0	0	0	256	160	368	188	94	11	1	0	0	0	0	0	0
14	256	384	448	192	96	20	46	23	11	1	0	0	0	0	0	0	0
15	160	112	556	24	12	2	261	130	65	0	0	0	0	0	0	0	0
16	20	14	7	3	1	0	288	144	72	0	0	0	0	0	0	0	0

Figure IV-5. *subimage-index-array* for Image 1.

3. Primary High-Level Function

This section contains only one function--**process_images**. It calls most of the functions in the program and produces status output as various functions are executing.

4. Primary Functions

Eight functions make up this section of my program. One of the functions calls three of the eight functions,

acting as the outer of three loops in the process. The functions are:

- **make_edge_list (num-edges-array num-edges size)**--this function returns a list of certain indices into the **num-edges-array**. These indices are the ones whose cells hold the **num-edges** in question (in this thesis, **num-edges** is either five or six). As discussed before, the cells of the **num-edges-array** represent the three-cell by three-cell blocks that have five or six edges in the appropriate gradient array.

- **find_match (subimage-edge-list orig-image-edge-list num-edges)**--the high-level function **process_images** passes control to this function, which calls **find_likely_match** first to look for a match between the two images and then function **check_results** to check to see if the match has been correctly found.

- **find_likely_match (subimage-edge-list orig-image-edge-list)**--The outer of three nested loops, this function loops through the **orig-image-edge-list** calling function **second_loop** until it finds a pattern in the **orig-image-edge-list** that matches the pattern in the **subimage-edge-list** or until it runs out of elements in the **orig-image-edge-list**. The subimage pattern is found by subtracting each pair of indices in the **subimage-edge-list** from the first pair of indices in the **subimage-edge-list**. This same pattern is looked for within the **orig-image-edge-list**. Note that the pattern is based on the first pair of indices in the **subimage-edge-list** and on some pair of indices in the **orig-image-edge-list**. Figure IV-6 shows graphically what a matching pattern looks like in the ***subimage-num-edges-array*** and the ***orig-image-num-edges-array***. If a matching pattern is found, **find_likely_match** calls the function **compare_unique_index_numbers**, which tries to find more evidence that a good match has been found. If a good match has been found, **find_likely_match** returns a list of the two pairs of indices (one from each **edge-list**) that produced the matching patterns. If a good match has not been found, **find_likely_match** continues to loop through **orig-image-edge-list** looking for another possible match.

- **second_loop (subimage-pattern orig-image-edge-list orig-image-pattern orig-key-x orig-key-y)**--The middle of three nested loops, **second_loop** loops through the subimage-pattern calling **third_loop** until it has looped

Figure IV-6. Matching Patterns of Five-Edge Blocks (in Bold Type).

completely through subimage-pattern or until it has exhausted the possibility of finding a matching pattern in the version of the **orig-image-edge-list** that it is working with.

- **third_loop (orig-image-edge-list orig-key-x orig-key-y orig-delta-x orig-delta-y sub-delta-x sub-delta-y)**--The innermost of the three loops, **third_loop** loops through yet another version of the **orig-image-edge-list** until it finds a pair of indices (**orig-delta-x** and **orig-delta-y**) in the list that, when subtracted from **orig-key-x** and **orig-key-y** (a pair of indices peeled off the front **orig-image-edge-list** by **find_likely_match**), produce the same values as **sub-delta-x** and **sub-delta-y** (a pair of numbers in the subimage-pattern) or until all pairs of indices in **orig-image-edge-list** have been examined. If there is a match, **third_loop** returns a list made up of **orig-delta-x** and **orig-delta-y.** If there is no match, it returns an empty list.

- **find_pattern (edge-list listlength)**--An auxiliary function, **find_pattern** finds a pattern that exists in **edge-list.** The pattern that it finds is the absolute value difference between the first pair of numbers in the list and every other pair of numbers in the list. For example, if the edge-list were:

  (1 2 5 6 3 8 7 9)

  then the pattern would be found as follows:

1	2	1	2	1	2
-5	-6	-3	-8	-7	-9
4	4	2	6	6	7

  and the resulting pattern would be:

  (4 4 2 6 6 7)

- **compare_unique_index_numbers (sub-image-x sub-image-y orig-image-x orig-image-y)**--This function looks for more evidence that the possible match is the correct match. It simply compares the value of ***subimage-index-array*** as indexed by **sub-image-x** and **sub-image-y** with the value of ***orig-image-index-array*** as indexed by **orig-image-x** and **orig-image-y.** If the values are the same, it returns a list whose contents are **sub-image-x**, **sub-image-y**, **orig-image-x**, and **orig-image-y.** If they are not the same, it returns nil. The function is called by function **find_likely_match.**

- **check_results (likely-match num-edges)**--Called by function **find_match**, **check_results** does just that--checks to see if the match found by find_match is indeed the correct match. This function will work only when the subimage is taken from the original image and therefore cannot be used in real applications.

5. Display Functions

The four functions in this portion of the program display the original image on the graphics screen. See section E for a more complete description of these functions.

6. Miscellaneous Functions

Two functions print arrays so that the user can inspect their contents easily; taking certain lists that the program sees as lists of independent elements but that are logically lists of pairs, two of the functions print these lists in their logical form for the benefit of the user; and two open and close the optional output file.

- **print_array (array-name size start-x start-y)**--This prints the contents of a square array onto the screen.

- **print_array_file (array-name size start-x start-y filename)**--Like **print_array** except that the output goes to a file instead of to the screen.

- **print_paired_list (list-name)**--This prints to the screen a list, such as (1 2 3 4), whose contents are logical pairs, as ((1 2)(3 4)), so that the user can better understand the list.

- **print_paired_list_to_output_file (list-name)**--Like function does the same as **print_paired_list** except that the output goes to the optional output file of the program.

- **open_output_file (filename)**--This enables the user to store the output of the program to a file.

- **close_output_file (filename)**--This closes the option output file opened by open_output_file.

H. ERROR CHECKING AND USER FRIENDLINESS

1. Error Checking

The only error checking done by the program is to make sure that the point in the original image chosen by the user to be the upper left-hand corner of the subimage will give a virtual subimage entirely inside the original image.

2. User Friendliness

The program was designed so that the user would have to call only a minimum number of functions manually. Status messages let the user know what processing is being done, so that the user is not staring at a blank screen wondering if anything is happening. The program also produces intermediate results that assist the user in analyzing the results of the run. Lastly, names and function names that made sense were used, and an attempt was made to write clear documentation for each function.

I. DIFFICULTIES WITH THRESHOLD

The thorniest problem in this thesis is that of finding a good threshold for the images. Without a good threshold, either too few or too many edges are produced, and either no matches are found or the system stack overflows during the list processing loops. Unfortunately, as can be seen in

Figures IV-7 and IV-8, the function relating the threshold to the number of five-edge cells is not monotonically increasing. This is because we are not looking simply at edges but at overlapping groups of edges found in the three-cell by three-cell blocks into which the two gradient arrays were logically divided. Since it was not possible easily to predict the behavior of the threshold, the only alternative was to limit the number of thresholds tested and pick the one that produced the best number of five-edge blocks in the subimage gradient array. In a real application, the goal would be to find the threshold that produces the best number of five-edge blocks in both images. The smaller the number of five-edge blocks, the faster the program runs; ideally, the number should be no more than 2.5% of the total number of pixels in the image. Better approaches remain a matter for future research.

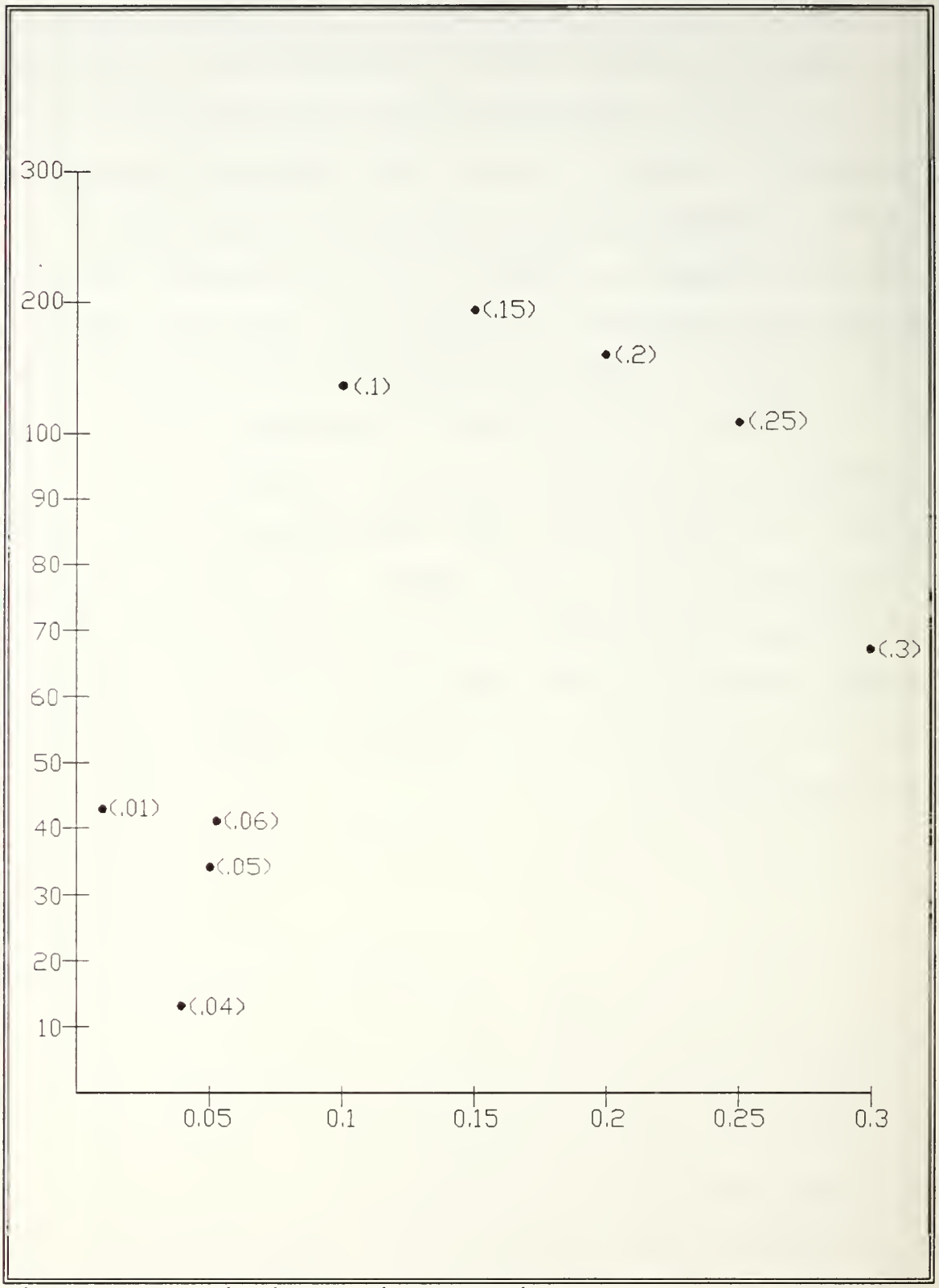

Figure IV-7. Difficulty in Selecting Threshold for Image 2.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	3	3	4	3	4	3	3	3	3	3	3	3	3	3	3
1	0	1	2	2	2	2	3	3	3	3	3	4	4	4	3	3	3
2	0	0	1	2	3	2	1	0	0	1	2	4	4	3	1	0	0
3	0	0	1	2	3	2	1	0	0	1	3	6	6	4	1	0	1
4	0	0	1	2	3	2	1	0	0	1	3	6	6	4	1	1	2
5	0	0	0	0	0	0	0	0	0	1	3	5	4	2	1	2	3
6	0	0	0	0	1	2	3	3	2	2	2	3	2	1	2	3	3
7	3	3	2	1	1	2	3	4	4	4	3	3	2	1	2	2	2
8	3	3	2	1	1	2	3	4	4	3	2	2	2	1	1	1	1

a. Partial ***subimage-num-edges-array*** with ***subimage-threshold*** of .35

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	4	4	5	6	7	6	5	4	3	3	4	4	5	5	6	5	4
1	4	6	7	7	6	4	3	3	3	3	4	5	7	7	8	6	5
2	3	4	5	6	6	4	3	3	3	3	4	5	7	6	6	4	3
3	3	4	4	5	6	6	6	6	5	4	4	6	8	6	4	2	3
4	1	1	1	2	4	5	7	7	6	4	4	6	7	5	2	2	4
5	0	0	0	0	1	2	4	4	4	3	4	6	6	4	2	2	5
6	0	1	2	3	3	3	4	4	5	4	5	5	5	3	3	5	7
7	3	4	5	5	4	3	3	4	6	6	6	5	6	5	5	4	5
8	5	6	6	7	5	5	5	6	7	5	5	4	6	5	4	3	4

b. Partial ***subimage-num-edges-array*** with ***subimage-threshold*** of .2

Figure IV-8. Effect of Threshold on Edges.

V. RESULTS

A. INTRODUCTION

The program produced better results than was thought possible. As can be seen in Figure V-1, as few as two pairs of indices in the ***subimage-five-edge-list*** or the ***subimage-six-edge-list*** were sufficient for the program to find the correct location of the subimage within the original image. It had been anticipated that possibly as many as five such pairs would be needed before the program could identify the registration of the subimage.

B. EXPERIMENTAL PROCEDURES

Two black and white photographs digitized to two 500 by 500 images were the basis for the experiments that were run with the program. The original photographs can be seen in their digitized form in Figures III-1 and III-2. Both of these images were reduced to forty by forty images and placed in files. Both Image 1 and Image 2 contain roads, trees, bushes, low ground cover, varying shades of dirt, and angles cause by meeting roads or borders. In addition, Image 1 contains a reservoir. Good representatives of relatively arid lands, both contain areas of relatively sharp distinction between light and dark. Such sharp distinctions may not be visible in heavily wooded areas or in flat, cultivated areas.

Threshold	*subimage-x*	*subimage-y*	Result	Number of 5-edge blocks in subimage	Number of 6-edge blocks in subimage
Image 1					
.35	10	20	match	8	0
.35	10	10	match	8	0
.3	8	17	match	9	0
.3	0	8	match	14	4
.3	18	18	match	3	0
Image 2					
.2	5	5	match	2	0
.175	3	8	match	8	3
.175	13	13	no match	0	0
.15	8	3	match	19	4
.15	20	0	match	33	19

Figure V-1. Sample Results of Running the Program Without Function select_function.

With both image files, experiments were conducted with histograms, manually choosing thresholds until fairly good thresholds for each image was found. Figure IV-8 shows some of the thresholds tried and the effect they had on the number of five-edge blocks found in one of the images.

After workable thresholds were found, the program was run eighteen times on each image, using various thresholds and values for ***subimage-x*** and ***subimage-y***. Figure V-1 shows representative results of these runs. Appendix B shows complete results of these tests.

Based on the results of the threshold work and of the test runs of the program, function **select_threshold** was written automatically to choose the threshold for the run based on the number of five-edge blocks the threshold produces in the ***subimage-gradient-array***. To test the program at this point, it was run on both images using every possible ***subimage-x***

and ***subimage-y**** value. The program produced correct results on all 800 tests conducted, thus fully proving the correctness of the program for these two images. Appendix B shows the results of twenty of these tests.

C. AVERAGE CPU TIMES

The tests which were run before function **select_threshold** was added were done on three different Symbolics computers, each one with different amounts of memory and different processing speeds. On the Symbolics 3675, with five megabytes of memory, it took an average of 70 seconds to run the program. The Symbolics 3640, with 1024 kilobytes of memory, ran the program in an average of 145 seconds, while the Symbolics 3650 with five megabytes of memory ran the program on the average in 85 seconds. Adding function **select_threshold** slowed the runs considerably. For instance, it took an average of 160.6 seconds to run the program on the Symbolics 3675.

A few things increased unnecessarily the CPU time needed to run the program. The program experiments with both five- and six-edge blocks to see if one has an advantage over the other. A real system would need only to use one or the other kind. Function **select_threshold** selects a threshold based on the number of five-edge blocks produced; this sometimes has the side effect of producing a large number of six-edge

blocks, which in turns adds to the processing time. A real system could avoid the extra processing by only processing one kind of block. Lastly, picking the threshold based on the number of five-edge blocks produced in the ***subimage-num-edges-array*** sometimes produces excessive number of five-edge blocks in the ***orig-image-num-edges-array***; thus, the artificiality of extracting the smaller subimage from within the original image caused disproportionate numbers of five-edge blocks in the original image. This would not occur with a real vision system, since it would compare two entirely separate and equally-sized images; most likely, the threshold chosen for one such image would produce a reasonable number of edge blocks in both.

Another way to speed up the program is to use parallel processing, particularly for the functions that cost the most in CPU time. Function **calculate_num_edges_and_unique_index_values** is a good candidate for parallel processing. What makes the function time-consuming is that it explores every possible three-cell by three-cell block in the appropriate gradient array; splitting the gradient array into, say, four pieces and assigning each piece to a separate processor to explore its three-cell by three-cell blocks would probably speed up the function significantly. Function **select_threshold** is another function whose CPU time could be reduced by parallel processing. This function is time

consuming because it calls functions **calculate_num_edges_and_unique_index_numbers** and **make_histogram** for every threshold it considers; to speed up the processing, the thresholds to explore could be divided amongst the processors available and the results compared later.

D. ACCURACY OF THE RESULTS

The program correctly found the registration of the subimage within the original image in 100% of the runs in which more than one pair of indices were in either the ***subimage-five-edge-list*** or in the ***subimage-six-edge-list***. In the runs in which there was one or zero pairs of indices in the subimage edge lists, the program correctly determined that it could not proceed. After the function **select_threshold** was added to the program, there were always more than one pair of indices at least in the ***subimage-five-edge-list***.

VI. CONCLUSIONS

A. MAJOR ACHIEVEMENTS

The program accurately located the same point in two images without comparing the images' gray levels pixel by pixel. It found the point by comparing the statistics about a few interesting points in each image.

The program demonstrated the practicability of adopting its techniques for picture registration determination. These techniques are:

- Computing the gradient for each pixel in the image.
- Logically dividing the gradient array into overlapping nine-cell blocks.
- Gathering statistics about the overlapping nine-cell blocks, specifically, the number of edges in each block and the position of those edges within the block.
- Studying only those blocks that contain an interesting number of edges--in the case of my thesis, those containing five and six edges.
- Looking for a pattern of five-edge blocks or six-edge blocks in the original image's **num-edges-array** that matches the pattern of five-edge blocks or six-edge blocks in the subimage's **num-edges-array.** This gives strong evidence that the first five- or six-edge block in each pattern represents the same point.
- Adding more evidence that the blocks found in the preceding paragraph represent the same point by seeing if the edges in those blocks are also in matching positions inside the blocks.

B. WEAKNESSES

Two of the functions take the lion's share of the processing time required to run the program. These are **select_threshold** and **calculate_num_edges_and_unique_index_numbers**, which may be called by **select_threshold** several times before a final threshold is chosen. Even on the fastest of the Symbolics computers available to me, **calculate_num_edges_and_unique_index_numbers** takes one minute of CPU time to process a forty by forty array.

The looping functions **find_likely_match** and **second_loop** do some redundant calculations in order to preserve the loop integrity. A clever algorithm needs to be written to avoid the redundancy. This program takes into account changes in height of the observer but not rotational movement of the observer. Rotational movement of the observer should be addressed by future thesis work.

C. OTHER CONCLUSIONS

The threshold should be the same for both images compared in a run of the program. If this is not the case, edges will not necessarily appear in identical spots in the two images and certainly not in the same patterns within identical nine-cell blocks. Finding the same spot in two images without some of the edges in identical places in the two images would be impossible using these techniques. Certainly, the

threshold will need to change to accommodate changing visibility.

Focusing the attention of this program on the nine-cell blocks that contained five and six edges is somewhat arbitrary. The program showed that either one of these types of blocks produces results if the threshold produces enough, but not too many, such blocks. Looking for both five-edges blocks and six-edge blocks is redundant and would not be needed in a practical application. These were chosen as the focus of the program because they appeared statistically interesting after some preliminary histogram work. However, three-edge blocks and seven-edge blocks are likely to be as eligible for use as the five- and six- edge blocks were. In fact, one possible way to reduce the time it takes to choose a threshold is to hold the threshold constant and select the number of edges of interest instead of selecting a threshold.

Testing this technique on two photographs taken moments apart remains to be done. This was not done during this study because such photographs were not readily available. Using successive photographs presents five problems: rotation of the observer, translation of the observer, changes in the height of the observer, changes in perspective due to the observer's translation, and changes in light. The first problem is not solved by the technique as presented in this thesis; however, it is possible that enlarging the size of the

blocks studied would ensure that some edges common in both images would appear in corresponding blocks in the two images and in the same relative positions within those blocks. Further research is needed into this area. The second and third problems can be handled by the program as it is currently written. As to the changes in perspective due to the translation of the observer, it is unlikely that the perfect results obtained in these experiments could be duplicated using two successive images; their common terrain will certainly look different to the computer, however slightly, because of the change in angle from the observer to the terrain as the observer traveled over it. As shown by the results of the experiments conducted in this study, it takes only a small number of common nine-cell blocks to appear the same in both images in order to obtain a match. Therefore, if only a few blocks in each image look the same to the computer, a match could be found; but whether even a few would appear the same in both primal sketches remains the work of future experiments. Changes in light is the last problem presented by analyzing two images taken at different times. The program's automatic threshold-selection function needs to be modified to choose a separate threshold for each image if there is a change in light. Acquiring a satisfactory threshold for each image does not guarantee that the thresholds will produce identical nine-cell blocks throughout

the two images; however, it should produce enough to find a match. Verifying this hypothesis remains the work of future experiments.

D. SUMMARY

The results of this thesis show that the techniques explored can be used successfully in photo interpretation, especially for motion detection. The techniques work very well on the reduced versions of Image 1 and Image 2, implying that they can be applied successfully to aerial photographs that do not contain highly regular, repeated patterns. They may work equally well on photographs that do contain such patterns, but that type of photograph was not available for testing. The techniques would certainly produce good results with any photograph that has sufficient changes in gray levels from pixel to pixel to produce detectable edges, including such photographs of the ocean bottom.

APPENDIX A

PROGRAM

A. CONVENTIONS USED

Throughout the program, the reader will find three typographical cues. The use of asterisks at the beginning and end of an expression signals a global variable (e.g., ***subimage-index-array***). Local variables (e.g., **num-edges**) and global variables have hyphens, while underscores within an expression and verbs identify a function name (e.g., **calculate_num_edges_and_unique_index_numbers** and **initialize**).

B. PROGRAM

The program is intended to be used in conjunction with a graphics terminal so that the user can display the images being processed. In order to use the program on a Symbolics machine that has no graphics terminal attached, the user must delete two of the display functions--**make_color_window** and **make_blue_window**--and two of the global variable declarations near those functions--**defvar *color-window*** and **defvar *my-window***.

The program processes forty-pixel by forty-pixel images, each stored in its own image file. The image files must be in the following format:

- Non-binary.
- The first item in the file must be the length of one of the sides of the image; this is so that the program can use any square image later on.
- The rest of the items must be the grey level values, from zero to one, of every pixel in the image, in row-major order from the upper left-hand corner of the image.The output is to the screen and, if desired, to an output file. The output consists of:

    1. Messages indicating the x and y coordinate chosen by the user and the status of the image preprocessing.

    2. ***subimage-five-edge-list***--a list of the array indices of the ***subimage-num-edges-array*** cells that represent the three-pixel by three-pixel blocks which contain five edges. The image represented is the subimage.

    3. ***subimage-six-edge-list***--a list of the array indices of those ***subimage-num-edges-array*** cells that represent the three-pixel by three-pixel blocks which contain six edges. Again, the image represented is the subimage.

    4. ***orig-image-five-edge-list***--a list of the array indices of those ***orig-image-num-edges-array*** cells that represent the three-pixel by three-pixel blocks which contain five edges. Here, the image represented is the original image.

    5. ***orig-image-six-edge-list***--a list of the array indices of those ***orig-image-num-edges-array*** cells that represent the three-pixel by three-pixel blocks which contain six edges. Again, the image represented is the original image.

    6. A message indicating:

        a. Whether the program found the correct x-y coordinate of the upper left-hand corner of the subimage in the original image; if it could not find the correct x-y coordinate, why it could not.

b. If the program found the correct x-y coordinate, whether the program compared five-edge blocks or six-edge blocks or both to do so.

Figure A-1 is an example of the screen output produced by the program. Actual file output is contained in Appendix C.

CAUTION: If the user uses function **open_output_file**, he must also use the function **close_output_file** at the end of the run or at the end of the session. If he does not, the file will not be closed, and the user will not be able to access it.

Figure A-1. Sample Screen and File Output.

```
;;; -*- Base: 10; Mode: LISP; Syntax: Common-lisp; Package: USER -*-

(defvar *main-screen*)
(defvar *size*)
(defvar *subimage-size*)
(defvar *fd*)
(defvar *fd2*)
(defvar *output-file-status*)
(defvar *test-status*)
(defvar *orig-image-threshold*)
(defvar *subimage-threshold*)
(defvar *subimage-x*)
(defvar *subimage-y*)
(defvar *orig-image-array*)
(defvar *orig-image-num-edges-array*)
(defvar *subimage-num-edges-array*)
(defvar *orig-image-index-array*)
(defvar *subimage-index-array*)
(defvar *orig-image-gradient-array*)
(defvar *subimage-gradient-array*)
(defvar *orig-image-edge-histogram*)
(defvar *subimage-edge-histogram*)
(defvar *orig-image-five-edge-list*)
(defvar *subimage-five-edge-list*)
(defvar *orig-image-six-edge-list*)
(defvar *subimage-six-edge-list*)

;;;
```

```
            (< yoffset 0))
          (pprint "Either x-offset or y-offset, or both, is out of range"))))

;*****************************************************************************************
; initializes all arrays.  Intended to be the only section that needs to be changed as the *
; problem changes.  Also initializes *subimage-size* and *output-file-status*.             *
;*****************************************************************************************
(defun initialize ()
  (setq *subimage-size* 20)
  (setq *output-file-status* 0)
  (setq *test-status* 0)
  (setq *orig-image-num-edges-array* (make-array (list (- *size* 3)(- *size* 3))))
  (setq *subimage-num-edges-array*
        (make-array (list (- *subimage-size* 3)(- *subimage-size* 3))))
  (setq *orig-image-gradient-array* (make-array (list (- *size* 1)(- *size* 1))))
  (setq *subimage-gradient-array*
        (make-array (list (- *subimage-size* 1)(- *subimage-size* 1))))
  (setq *orig-image-edge-histogram* '())
  (setq *subimage-edge-histogram* '())
  (setq *orig-image-index-array* (make-array (list (- *size* 3)(- *size* 3))))
  (setq *subimage-index-array* (make-array (list (- *subimage-size* 3)(- *subimage-size* 3))))
  (setq *orig-image-five-edge-list* '())
  (setq *orig-image-six-edge-list* '())
  (setq *subimage-five-edge-list* '())
  (setq *subimage-six-edge-list* '()))


;#########################################################################################
;                                  preprocessing functions                               #
;#########################################################################################


;*****************************************************************************************
; This function calculates the gradient.  Parameters are the name of the image array, the *
; length of one of its sides assuming the array is square, and the name of the array into *
; which the results of the calculations is to be stored.  The latter array will be smaller*
; than the image array by 1 in both dimensions.  Fills the array with the gradient       *
; calculated at each point.  Gradient is the square root of the sum of the square of the  *
; difference between the gray levels of the point and its right neighbor and of the       *
; square of the difference between the gray levels of the gray levels of the point and    *
; its neighbor immediately below.  Called by function process_images.                     *
;*****************************************************************************************
(defun calculate_gradient (size x y gradient-array)
   (dotimes (i (- size 1))
      (dotimes (j (- size 1))
         (setf (aref gradient-array j i)
               (sqrt (+ (expt (- (aref *orig-image-array* (+ j x)(+ i y))
                                 (aref *orig-image-array* (+ j x)(+ (+ i 1) y))) 2)
                        (expt (- (aref *orig-image-array* (+ j x)(+ i y))
                                 (aref *orig-image-array* (+ (+ j 1) x) (+ i y))) 2)))))))

;*****************************************************************************************
; This function looks at at the overlapping 9-cell blocks of an array, starting at the    *
; array's upper left-hand corner.  There is a 9-cell block at each point; the members of the *
; 9-cell block are the point itself, its two neighbors to its right, the neighbor         *
; immediatly beneath it and this neighbor's two neighbors to the right, and the three     *
; neighbors immediately below those three neighbors.  Each cell in the 9-block array that  *
; exceeds a given threshold is given a value of 2 raised to x, where x is a number between *
; 0 and 8 and equates to the ordinal value of the cell if the cells are numbered from 0 to 8 *
; from the upper left hand corner.  Last, all these values are added up and the sum is     *
; then used to represent the 9-cell block in an array whose indices also signify the upper *
; left hand corner coordinates of the 9-cell block in the original array.                  *
;*****************************************************************************************
(defun calculate_num_edges_and_unique_index_numbers
       (gradient-array size threshold unique-index-num-array num-edges-array)
   (let ((temp-array nil)(sum nil)(num-edges nil))
      (setq temp-array (make-array '(3 3)))
      (dotimes (i (- size 3))
         (dotimes (j (- size 3))
            (setq sum 0)
            (setq num-edges 0)
            (dotimes (k 3)
               (dotimes (l 3)
                  (cond ((or (= (aref gradient-array (+ l j)(+ k i)) threshold)
```

```
                        (> (aref gradient-array (+ 1 j)(+ k i)) threshold))
                    (setf (aref temp-array 1 k) 1))
                   ((< (aref gradient-array (+ 1 j)(+ k i)) threshold)
                    (setf (aref temp-array 1 k) 0)))
              (cond ((= (aref temp-array 1 k) 1)
                     (setq sum (+ sum (expt 2 (+ 1 (* 3 k)))))
                     (setq num-edges (+ num-edges 1))))))
        (setf (aref unique-index-num-array j i) sum)
        (setf (aref num-edges-array j i) num-edges)))))

;*****************************************************************************************
; This function selects the threshold to be used throughout a particular run, based on the  *
; number of five-edge blocks there are in the subimage's gradient array.  The number of     *
; edges found throughout the subimage's gradient array are stored in *subimage-num-edges-   *
; array.  If there are not at least 10 five-edge blocks available with the first threshold  *
; tried, another threshold is selected.  Each time a threshold is selected, calculate_num_  *
; edges_and_unique_index_numbers must be run and an edge-histogram created based on the     *
; the results.  This process continues until a threshold produces sufficient number of     *
; five-edge blocks.  The thresholds available to this function are given it in a list called*
; nine-best-list; it contains (.1 .2 .3 .4 .5 .6 .7 .8 .9).  The function attempts to       *
; bracket the goal between the number of five-edge blocks produced by successive thresholds.*
; Once this happens, it tries to get a little closer to the goal by trying either .05 above *
; or .05 below the best threshold so far.  Which ever of these produces five-edge blocks    *
; closest to the goal is the threshold used elsewhere in the program to produce the         *
; *orig-num-edges-array*, *subimage-num-edges-array*, *orig-image-index-array*, and the     *
; *subimage-index-array*.                                                                   *
;*****************************************************************************************
(defun select_threshold ()
  (let ((lower-bracket 1000)(upper-bracket 0)(goal 10)(nine-best-list nil)
        (edge-histogram nil)(test-threshold nil)(temp-threshold nil))
    (setq nine-best-list '(.1 .2 .3 .4 .5 .6 .7 .8 .9))
    (do ((best-list nine-best-list (cdr best-list)))
        ((and (< goal upper-bracket)
              (>= goal lower-bracket)))
      (setq temp-threshold (car best-list))
      (setq upper-bracket lower-bracket)
      (calculate_num_edges_and_unique_index_numbers *subimage-gradient-array*
                              *subimage-size* temp-threshold *subimage-index-array*
                              *subimage-num-edges-array*)(princ " ."))
      (setq edge-histogram (make_histogram *subimage-num-edges-array* *subimage-size*))
      (setq lower-bracket (cadddr (cddr edge-histogram))))
    (terpri)
    (cond ((= lower-bracket goal) temp-threshold)
          ((and (= (abs (- goal lower-bracket))
                   (abs (- goal upper-bracket)))
                (<= lower-bracket upper-bracket)
                (> lower-bracket 1)) temp-threshold)
          ((or (and (= (abs (- goal lower-bracket))
                       (abs (- goal upper-bracket)))
                    (>= lower-bracket upper-bracket))
               (and (= (abs (- goal lower-bracket))
                       (abs (- goal upper-bracket)))
                    (<= lower-bracket upper-bracket)
                    (< lower-bracket 2)))(- temp-threshold .1))
          ((and (< (abs (- goal lower-bracket))
                   (abs (- goal upper-bracket)))
                (/= lower-bracket goal))
           (setq test-threshold (- temp-threshold .05))
           (calculate_num_edges_and_unique_index_numbers *subimage-gradient-array*
                              *subimage-size* test-threshold *subimage-index-array*
                              *subimage-num-edges-array*)
           (setq edge-histogram (make_histogram *subimage-num-edges-array* *subimage-size*))
           (cond ((< (abs (- goal (cadddr (cddr edge-histogram))))
                     (abs (- goal lower-bracket))) test-threshold)
                 ((and (= (abs (- goal (cadddr (cddr edge-histogram))))
                          (abs (- goal lower-bracket)))
                       (<= (cadddr (cddr edge-histogram)) lower-bracket)
                       (> (cadddr (cddr edge-histogram)) 1)) test-threshold)
                 ((or (and (= (abs (- goal (cadddr (cddr edge-histogram))))
                              (abs (- goal lower-bracket)))
                           (>= (cadddr (cddr edge-histogram)) lower-bracket))
                      (and (= (abs (- goal (cadddr (cddr edge-histogram))))
                              (abs (- goal lower-bracket)))
                           (<= (cadddr (cddr edge-histogram)) lower-bracket)
                           (< (cadddr (cddr edge-histogram)) 2))) temp-threshold)
                 ((> (abs (- goal (cadddr (cddr edge-histogram))))
```

```
                (abs (- goal lower-bracket))) temp-threshold)))
        ((and (> (abs (- goal lower-bracket))
                 (abs (- goal upper-bracket)))
              (/= lower-bracket goal))
         (setq temp-threshold (- temp-threshold .1))
         (setq lower-bracket upper-bracket)
         (calculate_num_edges_and_unique_index_numbers *subimage-gradient-array*
                  *subimage-size* temp-threshold *subimage-index-array*
                  *subimage-num-edges-array*)
         (setq test-threshold (+ temp-threshold .05))
         (calculate_num_edges_and_unique_index_numbers *subimage-gradient-array*
                           *subimage-size* test-threshold *subimage-index-array*
                           *subimage-num-edges-array*)
         (setq edge-histogram (make_histogram *subimage-num-edges-array* *subimage-size*))
         (cond ((< (abs (- goal (cadddr (cddr edge-histogram))))
                   (abs (- goal lower-bracket))) test-threshold)
               ((and (= (abs (- goal (cadddr (cddr edge-histogram))))
                        (abs (- goal lower-bracket)))
                     (<= (cadddr (cddr edge-histogram)) lower-bracket)
                     (> (cadddr (cddr edge-histogram)) 1)) test-threshold)
               ((or (and (= (abs (- goal (cadddr (cddr edge-histogram))))
                            (abs (- goal lower-bracket)))
                         (>= (cadddr (cddr edge-histogram)) lower-bracket))
                    (and (= (abs (- goal (cadddr (cddr edge-histogram))))
                            (abs (- goal lower-bracket)))
                         (<= (cadddr (cddr edge-histogram)) lower-bracket)
                         (< (cadddr (cddr edge-histogram)) 1))) temp-threshold)
               ((>= (abs (- goal (cadddr (cddr edge-histogram))))
                    (abs (- goal lower-bracket))) temp-threshold))))))
```

```
;*****************************************************************************************
; This function creates a histogram in which is stored the number of edges that are found  *
; within each of the nine-cell blocks, using num-edges-array as the provider of the        *
; statistics.  Num-edges-array is filled by function calculate_num_edges_and_unique_index_ *
; numbers.  Inputs to this function are:  the array holding the edge information for the    *
; appropriate image, the name of the list in which to store the edge count, and the size of *
; the image (i.e., the length of one of its sides).                                         *
; Since there are 10 possible counts of edges in each nine-cell block, an array is not a    *
; convenient data structure.  Instead, the count is stored in a list whose values represent *
; the number of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 edges, respectively, found in all the nine-cell*
; blocks of the image.                                                                      *
; ****************************************************************************************
(defun make_histogram (num-edges-array size)
  (let ((sum0 0)(sum1 0)(sum2 0)(sum3 0)(sum4 0)(sum5 0)(sum6 0)(sum7 0)(sum8 0)(sum9 0)
        (edge-histogram nil))
  (dotimes (i (- size 3))
    (dotimes (j (- size 3))
      (cond ((= 0 (aref num-edges-array j i))
             (setq sum0 (+ sum0 1)))
            ((= 1 (aref num-edges-array j i))
             (setq sum1 (+ sum1 1)))
            ((= 2 (aref num-edges-array j i))
             (setq sum2 (+ sum2 1)))
            ((= 3 (aref num-edges-array j i))
             (setq sum3 (+ sum3 1)))
            ((= 4 (aref num-edges-array j i))
             (setq sum4 (+ sum4 1)))
            ((= 5 (aref num-edges-array j i))
             (setq sum5 (+ sum5 1)))
            ((= 6 (aref num-edges-array j i))
             (setq sum6 (+ sum6 1)))
            ((= 7 (aref num-edges-array j i))
             (setq sum7 (+ sum7 1)))
            ((= 8 (aref num-edges-array j i))
             (setq sum8 (+ sum8 1)))
            ((= 9 (aref num-edges-array j i))
             (setq sum9 (+ sum9 1))))))
  (setq edge-histogram (list sum0 sum1 sum2 sum3 sum4 sum5 sum6 sum7 sum8 sum9))))

;#########################################################################################
;                                  primary functions                                     #
```

```
;*****************************************************************************************
; This function creates a list of x-y coordinates that correspond to the upper left-hand cell*
; of particular nine-cell blocks in an image.  The function looks through the array that     *
; holds the number of edges found in each nine-cell block of the image and puts in the list  *
; a copy of the x-y coordinates that correspond to the upper left-hand cells of the nine-cell*
; blocks that have num-edges number of edges.  Inputs are the name of the array that holds   *
; the number-of-edges information about the image in question, the number of edges of        *
; interest, the size of the image, and name of the list into which to store the results of   *
; the function.                                                                              *
;*****************************************************************************************
(defun make_edge_list (num-edges-array num-edges size)
  (let ((list-name nil))
    (dotimes (i (- size 3) list-name)
      (dotimes (j (- size 3) list-name)
        (cond ((= num-edges (aref num-edges-array j i))
               (setq list-name (append list-name (list j i)))))))))

;*****************************************************************************************
; This function is the high-level function that calls three nested loops into action.       *
; The result of the nested loops is stored in likely-match, which represents the first point *
; in the original image that the program thinks might be a match to the point of interest    *
; in the subimage.  This match is based on finding the same number of edges in the nine-cell *
; block defined by the first pair of array indices in the subimage-edge-list as in the nine- *
; cell block defined by the likely-matching point in the original image; and is based on the *
; finding the same pattern of five-edge blocks starting at the point defined by the first    *
; pair of array indices in the subimage-edge-list as the pattern of five-edge blocks starting*
; at the likely-matching point in the original image; and finally is based on these two      *
; haveing the same index numbers assigned them in their index-arrays.  likely-match may be  *
; nil, in which case there is no hope to find a match in the two images.  If likely-match   *
; is not nil, check_results is called.  Inputs are the either *subimage-five-edge-list* or   *
; *subimage-six-edge-list, *orig-image-five-edge-list* or *orig-image-six-edge-list*, and    *
; the corresponding number of edges, either 5 or 6.                                         *
;*****************************************************************************************
(defun find_match (subimage-edge-list orig-image-edge-list num-edges)
  (let ((likely-match '()))
    (setq likely-match (find_likely_match subimage-edge-list
                                          orig-image-edge-list))
    (cond ((eq likely-match nil)
           (terpri)
           (princ "There are no matches in the edge lists")
           (terpri)
           (cond ((= *output-file-status* 1)
                  (terpri *fd*)
                  (princ "There are no matches in the edge lists" *fd*)
                  (terpri *fd*))))
          ((neq likely-match nil)
           (check_results likely-match num-edges)))))

;*****************************************************************************************
; This function is the outer of three loops.  It loops through the orig-image-edge-list until*
; it finds a pattern in the list that matches the pattern found in the subimage-edge-list.   *
; If a matching pattern is found, it sets likely-match-list to the result of the function    *
; compare_unique_index_numbers.  If no matching pattern is found, it calls itself with the   *
; orig-image-edge-list as it was when the first matching pattern was found.  If, in the end  *
; likely-match-list is nil, it again calls itself with the orig-image-edge-list in its       *
; latest state.  The function returns likely-match-list, as it eventually exists.            *
; Inputs are either *subimage-five-edge-list* or *subimage-six-edge-list* and either *orig-  *
; image-five-edge-list* or *orig-image-six-edge-list*.  Output is likely-match-list.  It     *
; calls functions second_loop, compare_unique_index_numbers, and itself.  It is called by    *
; function find_match.                                                                       *
;*****************************************************************************************
(defun find_likely_match (subimage-edge-list orig-image-edge-list)
  (let ((subimage-pattern nil)(orig-image-pattern nil)(likely-match-list '())
        (orig-key-x nil)(orig-key-y nil))
    (setq subimage-pattern
          (find_pattern subimage-edge-list (length subimage-edge-list)))
    (setq orig-image-pattern
      (do ((orig-image-edge-list-temp orig-image-edge-list (cddr orig-image-edge-list-temp)))
          ((or (tree-equal subimage-pattern orig-image-pattern)
               (< (length orig-image-edge-list-temp)(- (length subimage-edge-list) 2)))
           (list orig-key-x orig-key-y orig-image-pattern))
        (setq orig-image-pattern '(0))
        (setq orig-key-x (car orig-image-edge-list-temp))
```

```
      (setq orig-key-y (cadr orig-image-edge-list-temp))
      (setq orig-image-pattern
            (second_loop subimage-pattern orig-image-edge-list-temp orig-image-pattern
                         orig-key-x orig-key-y))
      (setq orig-image-edge-list (cddr orig-image-edge-list-temp))))
  (setq orig-key-x (car orig-image-pattern))
  (setq orig-key-y (cadr orig-image-pattern))
  (setq orig-image-pattern (caddr orig-image-pattern));(pprint orig-image-pattern)
  (cond ((tree-equal subimage-pattern orig-image-pattern)
         (setq likely-match-list
             (compare_unique_index_numbers (car subimage-edge-list)(cadr subimage-edge-list)
                          orig-key-x orig-key-y)) likely-match-list)
        (t (find_likely_match subimage-edge-list orig-image-edge-list)))
   (cond ((eq likely-match-list nil)
          (setq likely-match-list
             (find_likely_match subimage-edge-list orig-image-edge-list)) likely-match-list)
         (t likely-match-list))))


;*****************************************************************************************
; This function loops through the subimage-pattern and calls third_loop to look for the same *
; pattern in the orig-image-edge-list sent it by find_likely_match.  Inputs are the       *
; subimage-pattern, a modified version of either *orig-image-five-edge-list* or *orig-image- *
; six-edge-list*, orig-image-pattern, and the index values that were car'd and cadr'd off   *
; the orig-image-edge-list as it exists in find_likely_match.  It is from these index values *
; that the pattern in the rest of the orig-image-edge-list is formed.  Output is orig-image- *
; pattern.  Calls third_loop.                                                               *
;*****************************************************************************************
(defun second_loop (subimage-pattern orig-image-edge-list orig-image-pattern orig-key-x
                    orig-key-y)
  (let ((sub-delta-x nil)(sub-delta-y nil)(orig-delta-x '0)(orig-delta-y '0)
        (temp-orig-image-edge-list '())(delta-list '()))
    (setq orig-image-edge-list (cddr orig-image-edge-list))
    (setq temp-orig-image-edge-list orig-image-edge-list)
    (do ((temp-subimage-pattern subimage-pattern (cddr temp-subimage-pattern)))
        ((or (null temp-subimage-pattern )
             (null orig-image-pattern)) (cdr orig-image-pattern))
      (setq sub-delta-x (car temp-subimage-pattern))
      (setq sub-delta-y (cadr temp-subimage-pattern))
      (setq delta-list
            (third_loop temp-orig-image-edge-list orig-key-x orig-koy-y orig-delta-x
                        orig-delta-y sub-delta-x sub-delta-y))
      (setq orig-image-pattern (append orig-image-pattern delta-list)))))

;*****************************************************************************************
; The innermost of the three loops, this function loops through the orig-image-edge-list    *
; that is passed to it by second_loop.  It subtracts the first item in the list from        *
; orig-key-x and assigns it to orig-delta-x; it subtracts the second item in the list from  *
; orig-key-y and assigns it to orig-dolta-y.  If orig-delta-x equals sub-delta-x and if      *
; orig-delta-y equals sub-delta-y, that means that the distance between two array cells in  *
; *orig-image-num-edges-array* is tho same as the distance between two array cells in the   *
; *subimag-num-edges-array* and that a matching pattern is emerging from the orig-image-edge-*
; list.  If they are equal, orig-delta-x and orig-delta-y are put into a list, and the      *
; function returns this list.  If they are not equal, the function returns the null list.  *
; Called by second_loop.                                                                    *
;*****************************************************************************************
(defun third_loop (orig-image-edge-list orig-key-x orig-key-y orig-delta-x orig-delta-y
                   sub-delta-x sub-delta-y)(let ((delta-list '()))
  (do ((temp-orig-image-edge-list orig-image-edge-list (cddr temp-orig-image-edge-list)))
      ((or (null temp-orig-image-edge-list)
           (and (= orig-delta-x sub-delta-x)
                (= orig-delta-y sub-delta-y))) delta-list)
    (setq orig-delta-x (abs (- orig-key-x (car temp-orig-image-edge-list))))
    (setq orig-delta-y (abs (- orig-key-y (cadr temp-orig-image-edge-list))))
    (setq delta-list (list orig-delta-x orig-delta-y)))
  (cond ((and (= orig-delta-x sub-delta-x)
              (= orig-delta-y sub-delta-y)) delta-list)
        ((or (/= orig-delta-x sub-delta-x)
             (/= orig-delta-y sub-delta-y)) '()))))

;*****************************************************************************************
; This function finds a pattern within a list by finding the difference between each pair   *
; and the first pair.  Inputs are either *subimage-five-edge-list* or *subimage-six-edge-   *
; list and the length of that list.  It returns the pattern found.                          *
;*****************************************************************************************
(defun find_pattern (edge-list  listlength)
```

```
  (let ((first-x nil)(first-y nil)(pattern nil))
    (setq first-x (car edge-list))
    (setq first-y (cadr edge-list))
    (dotimes (i (- (/ listlength 2) 1) pattern)
      (setq edge-list (cddr edge-list))
      (setq pattern (append pattern (list (abs (- first-x (car edge-list)))
                                          (abs (- first-y (cadr edge-list))))))))))

;*****************************************************************************************
; This function checks the common edge list areas against the unique number for the     *
; corresponding nine-cell blocks.  Those nine-cell blocks with both matching number of edges *
; and matching unique numbers are then put into a list of lists.  Each list within the list *
; is made up the x-y coordinate of the subimage nine-cell block and then the x-y coordinate *
; of the original image nine-cell block.  No inputs are required.                       *
;*****************************************************************************************
(defun compare_unique_index_numbers (sub-image-x sub-image-y orig-image-x orig-image-y)
  (cond ((= (aref *subimage-index-array* sub-image-x sub-image-y)
            (aref *orig-image-index-array* orig-image-x orig-image-y))
         (list sub-image-x sub-image-y orig-image-x orig-image-y))
        (t nil)))



;*****************************************************************************************
; This function checks the program's guess as to the position of the subimage within the  *
; original image, i.e., compares the x-y coordinate of the upper left-hand corner of the  *
; subimage found by the program (and stored in *likely-match-list*) to the actual left-hand *
; corner of the subimage stored in the global variables *subimage-x* and *subimage-y*.    *
;*****************************************************************************************
(defun check_results (likely-match num-edges)
  (let ((stars '**********************************************************************))
    (cond ((null likely-match)
           (terpri)
           (princ "The program has not found the coordinates that correspond to the upper")
           (terpri)
           (princ "left-hand corner of the subimage.")
           (terpri)
           (pprint stars)
           (cond ((= *output-file-status* 1)
                  (terpri *fd*)
                  (princ "The program has not found the coordinates that correspond to the
                                  upper" *fd*)(terpri *fd*)
                  (princ "left-hand corner of the subimage." *fd*)
                  (terpri)
                  (pprint stars *fd*))))
          (t (cond ((and (= *subimage-x* (- (caddr likely-match)(car likely-match)))
                         (= *subimage-y* (- (cadddr likely-match)(cadr likely-match))))
                    (terpri)(terpri)
                    (princ "By comparing nine-block cells with ")
                    (princ num-edges)
                    (princ " edges, the program has correctly found ")
                    (terpri)
                    (princ "that the upper left-hand corner ")
                    (princ "of the subimage corresponds to the following ")
                    (terpri)
                    (princ "x and y coordinates, respectively, of the original image:")
                    (terpri)
                    (terpri)
                    (princ "                                          ")
                    (princ *subimage-x*)
                    (terpri)
                    (princ "                                          ")
                    (princ *subimage-y*)
                    (terpri)
                    (pprint stars)
                    (cond ((= num-edges 6)
                           (pprint stars)))
                    (cond ((= *output-file-status* 1)
                           (terpri *fd*)(terpri *fd*)
                           (princ "By comparing nine-block cells with " *fd*)
                           (princ num-edges *fd*)
                           (princ " edges, the program has correctly found" *fd*)
                           (terpri *fd*)
                           (princ "that the upper left-hand corner " *fd*)
                           (princ "of the subimage corresponds to the following " *fd*)
                           (terpri *fd*)
```

```
                        (princ "x and y coordinates, respectively, of the original image:"
                               *fd*)
                        (terpri *fd*)
                        (terpri *fd*)
                        (princ "                                          "*fd*)
                        (princ *subimage-x* *fd*)
                        (terpri *fd* )
                        (princ "                                          " *fd*)
                        (princ *subimage-y* *fd*)
                        (terpri *fd*)
                        (pprint stars *fd*)
                        (cond ((= num-edges 6)
                               (pprint stars *fd*)))))
                (cond ((= *test-status* 1)
                        (test_results num-edges)))))))))


;#########################################################################################
;                       primary high-level functions                                    #
;#########################################################################################

;*****************************************************************************************
; This function calls the other preprocessing functions and decides which are the most  *
; promising areas revealed by the preprocessing.                                        *
;*****************************************************************************************
(defun process_images ()
 (let ((stars '*********************************************************************************))
  (terpri)
  (princ *subimage-x*)
  (princ " and ")
  (princ *subimage-y*)
  (princ " are the x and y coordinates of the original image point")
  (terpri)
  (princ "that is the upper left-hand corner of the subimage.")
  (terpri)(terpri)
  (princ "running calculate_gradient on the original image")(terpri)
  (calculate_gradient *size* 0 0 *orig-image-gradient-array*)
  (princ "running calculate_gradient on the subimage")(terpri)
  (calculate_gradient *subimage-size* *subimage-x* *subimage-y* *subimage-gradient-array*)
  (princ "running select_threshold")
  (setq *subimage-threshold* (select_threshold))
  (setq *orig-image-threshold* *subimage-threshold*)
  (princ "running calculate_num_edges_and_unique_index_numbers on the original image")(terpri)
  (calculate_num_edges_and_unique_index_numbers
       *orig-image-gradient-array* *size* *orig-image-threshold*
                  *orig-image-index-array* *orig-image-num-edges-array*)
  (princ "running calculate_num_edges_and_unique_index_numbers on the subimage")(terpri)
  (calculate_num_edges_and_unique_index_numbers
    *subimage-gradient-array* *subimage-size* *subimage-threshold* *subimage-index-array*
      *subimage-num-edges-array*)
  (terpri)
  (princ "The threshold for this run is ")
  (princ *subimage-threshold*)(princ ".")
  (terpri)(terpri)
  (princ "*subimage-five-edge-list*:")(terpri)
  (setq *subimage-five-edge-list*
        (make_edge_list *subimage-num-edges-array* 5 *subimage-size*))
  (print_paired_list *subimage-five-edge-list*)(terpri)(terpri)
  (princ "*subimage-six-edge-list*:")(terpri)
  (setq *subimage-six-edge-list*
        (make_edge_list *subimage-num-edges-array* 6 *subimage-size*))
  (print_paired_list *subimage-six-edge-list*)(terpri)(terpri)
  (princ "*orig-image-five-edge-list*:")(terpri)
  (setq *orig-image-five-edge-list* (make_edge_list *orig-image-num-edges-array* 5 *size*))
  (print_paired_list *orig-image-five-edge-list*)(terpri)(terpri)
  (princ "*orig-image-six-edge-list*:")(terpri)
  (setq *orig-image-six-edge-list* (make_edge_list *orig-image-num-edges-array* 6 *size*))
  (print_paired_list *orig-image-six-edge-list*)(terpri)
  (pprint stars)
  (cond ((= *output-file-status* 1)
         (terpri *fd*)(terpri *fd*)
         (princ *subimage-x* *fd*)
         (princ " and " *fd*)
         (princ *subimage-y* *fd*)
         (princ " are the x and y coordinates of the original image point" *fd*)
         (terpri *fd*)
```

```
        (princ "that is the upper left-hand corner of the subimage." *fd*)
        (terpri *fd*)(terpri *fd*)
        (princ "running calculate_gradient on the orig-image" *fd*)(terpri *fd*)
        (princ "running calculate_gradient on the subimage" *fd*)(terpri *fd*)
        (princ "running select_threshold" *fd*)(terpri *fd*)
        (princ "running calcualte_num_edges_and_unique_index_numbers on the original image" *fd*)
        (terpri *fd*)
        (princ "running calculate_num_edges_and_unique_index_numbers on the subimage" *fd*)
        (terpri *fd*)(terpri *fd*)
        (princ "The threshold for this run is " *fd*)
        (princ *subimage-threshold* *fd*)
        (princ "." *fd*)
        (terpri *fd*)(terpri *fd*)
        (princ "*subimage-five-edge-list*:" *fd*)(terpri *fd*)
        (print_paired_list_to_output_file *subimage-five-edge-list*)
        (terpri *fd*)(terpri *fd*)
        (princ "*subimage-six-edge-list*:" *fd*)(terpri *fd*)
        (print_paired_list_to_output_file *subimage-six-edge-list*)
        (terpri *fd*)(terpri *fd*)
        (princ "*orig-image-five-edge-list*:" *fd*)(terpri *fd*)
        (print_paired_list_to_output_file *orig-image-five-edge-list*)
        (terpri *fd*)(terpri *fd*)
        (princ "*orig-image-six-edge-list*:" *fd*)(terpri *fd*)
        (print_paired_list_to_output_file *orig-image-six-edge-list*)(terpri *fd*)
        (pprint stars *fd*)(terpri *fd*)))
 (cond ((> (length *subimage-five-edge-list*) 2)
        (find_match *subimage-five-edge-list* *orig-image-five-edge-list* 5))
       ((eq *subimage-five-edge-list* '())
        (terpri)(terpri)
        (princ "There are no five-edge blocks in the subimage.")
        (terpri)
        (pprint stars)
        (cond ((= *output-file-status* 1)
               (terpri *fd*)(terpri *fd*)
               (princ "There are no five-edge blocks in the subimage." *fd*)
               (terpri *fd*)
               (pprint stars *fd*))))
       ((= 2 (length *subimage-five-edge-list*))
        (terpri)(terpri)
        (princ "There is only one five-edge block in the subimage.")
        (terpri)
        (pprint stars)
        (cond ((= *output-file-status* 1)
               (terpri *fd*)(terpri *fd*)
               (princ "There is only one five-edge blocks in the subimage." *fd*)
               (terpri *fd*)
               (pprint stars *fd*)))))
 (cond ((> (length *subimage-six-edge-list*) 2)
        (find_match *subimage-six-edge-list* *orig-image-six-edge-list* 6))
       ((eq *subimage-six-edge-list* '())
        (terpri)(terpri)
        (princ "There are no six-edge blocks in the subimage.")
        (terpri)
        (pprint stars)(pprint stars)
        (cond ((= *output-file-status* 1)
               (terpri *fd*)(terpri *fd*)
               (princ "There are no six-edge blocks in the subimage." *fd*)
               (terpri *fd*)
               (pprint stars *fd*)(pprint stars *fd*))))
       ((= 2 (length *subimage-six-edge-list*))
        (terpri)(terpri)
        (princ "There is only one six-edge block in the subimage.")
        (pprint stars)(pprint stars)
        (cond ((= *output-file-status* 1)
               (terpri *fd*)(terpri *fd*)
               (princ "There is only one six-edge block in the subimage." *fd*)
               (pprint stars *fd*)(pprint stars *fd*)))))))

;!!!
```

```
;*****************************************************************************
(defun make_color_window
       (&rest options &key (superior (color:find-color-screen : create-p t))
       &allow-other-keys)
  (apply #'tv:make-window 'tv:window
         :blinker-p nil
         :borders 2
         :save-bits nil
         :expose-p t
         :default-character-style
           '(:fix :bold :large)
         :label "IMAGE ANALYSIS"
         :superior superior
         options))

(defvar *color-window* (make_color_window))

;*****************************************************************************
; make_blue_window creates a blue window on the color graphics display   *
;*****************************************************************************
(defun make_blue_window (&rest options)
   (setf *main-screen* (send *terminal-io* :superior))
   (apply #'make_color_window
          :erase-aluf(send color:color-screen
                           :compute-color-alu
                           tv:alu-seta 0 0 0.5)
          options))

(defvar *my-window* (make_blue_window))


;*************************************************************************************************
; color-pixel colors a pixel with the inputted x and y coordinates in the color indicated by*
; the inputted shade                                                                            *
;*************************************************************************************************
(defun color_pixel (xcoord ycoord shade)
   (send *my-window* :draw-point xcoord ycoord shade))

;*****************************************************************************
; display_image draws the picture that is represented by the array    *
;*****************************************************************************
(defun display_image (array-name x y size)
   (let ((b&w-color nil))
   (dotimes (row-number size nil)
      (dotimes (col-number size nil)
         (setq b&w-color (aref array-name col-number row-number))
         (color_pixel (+ x col-number) (+ y row-number) (send (send *my-window* :screen)
                      :compute-color-alu color:alu-x b&w-color b&w-color b&w-color))))))


;#############################################################################################
;                          miscellaneous                                                    #
;#############################################################################################

;*************************************************************************************************
; With this function, the user can print an array to the screen.  Inputs are the array name,*
; the length of one of its sides (it must be square), and the indices into the array for    *
; the point at which the user wants the display to begin.                                   *
;*************************************************************************************************
(defun print_array (array-name size start-x start-y)
  (dotimes (i size)
    (terpri)
    (dotimes (j size)
      (princ (aref array-name (+ start-x j)(+ start-y i)))
      (princ " "))))

;*************************************************************************************************
; With this array, the user can print an array to a file.  The inputs are the same as to    *
; function print_array except for an extra one, which is the name, in double quotes, of the *
; in which to store the array.                                                              *
;*************************************************************************************************
(defun print_array_file (array-name size start-x start-y filename)
  (let ((fd nil))
    (setq fd (open filename :direction :output))
    (dotimes (i size)
```

```
      (terpri fd)
      (dotimes (j size)
        (princ (aref array-name (+ start-x j)(+ start-y i)) fd)
        (princ " " fd)))
    (close fd)))

;*****************************************************************************************
; This function does a pretty print of a list that is logically a list of paired lists, so*
; that the user can more readily understand the list's contents.  Input is the list's     *
; name.  Called by function process_images.                                               *
;*****************************************************************************************
(defun print_paired_list (list-name)
  (let ((paired-list '()))
  (dotimes (i (/ (length list-name) 2))
    (setq paired-list (append paired-list (list (list (car list-name)(cadr list-name)))))
    (setq list-name (cddr list-name)))
  (princ paired-list)))

;*****************************************************************************************
; This function does a pretty print of a list to the user's output file.  Called by       *
; process_images.                                                                         *
;*****************************************************************************************
(defun print_paired_list_to_output_file (list-name)
  (let ((paired-list '()))
  (dotimes (i (/ (length list-name) 2))
    (setq paired-list (append paired-list (list (list (car list-name)(cadr list-name)))))
    (setq list-name (cddr list-name)))
  (princ paired-list *fd*)))

;*****************************************************************************************
; This functions enables the user to store the output of the program to a file.  Called   *
; by the user, it has one input, the filename of the file in which the user wants to store *
; the output.  If the user uses this function, he must also use function close_output_file*
; before logging out, or the output file will remain open and inaccesible.  Setting       *
; *output-file-status* to 1 lets the functions that produce output know to send output to *
; to the output file.  Only one file may be open during any one run of process_images.    *
;*****************************************************************************************
(defun open_output_file (filename)
  (setq *fd* (open filename :direction :output))
  (setq *output-file-status* 1))

;*****************************************************************************************
; This function closes the output file.  Called by the user, it can be used at the end of *
; a run or at the end of the session.  Input is *fd*, the variable that takes on the      *
; the name of the output file.                                                            *
;*****************************************************************************************
(defun close_output_file (filename)
  (close filename)
  (setq *output-file-status* 0))


;*****************************************************************************************
; These functions test the program on every possible point in the original image.         *
; complete_test is called by the user; x and y are the coordinates where the test is to   *
; begin, ideally 0 and 0.  test_results is called by function check results, that is, it  *
; called only if the program succeeds in finding the correct coordinates of the upper     *
; left-hand corner of the subimage.                                                       *
;*****************************************************************************************
(defun complete_test (input-filename x y output-filename)
  (setq *fd2* (open output-filename :direction :output))
  (setq *test-status* 1)
  (make_image_array input-filename)
  (initialize)
  (do ((i y (+ i 1)))
      ((= i 21))
    (do ((j x (+ j 1)))
        ((= j 21))
      (set_up j i)(terpri *fd2*)
      (princ "subimage-x*: " *fd2*)(princ j *fd2*)(princ " *subimage-y*: " *fd2*)
      (princ i *fd2*)
      (process_images)))
  (setq *test-status* 0)
  (close *fd2*))

(defun test_results (num-edges)
  (terpri *fd2*)
```

```
(princ "The program found *subimage-x* to be " *fd2*)(princ *subimage-x* *fd2*)
(princ " and *subimage-y* to be " *fd2*)(princ *subimage-y* *fd2*)(princ " using " *fd2*)
(princ num-edges *fd2*)(princ " edge blocks. " *fd2*)
(princ "Threshold was " *fd2*)(princ *subimage-threshold* *fd2*))
```

APPENDIX B

RESULTS OF SELECTED TESTS

Without function select_threshold:

Threshold	*subimage-x*	*subimage-y*	Result	Number of 5-edge blocks in subimage	Number of 6-edge blocks in subimage
Image 1					
.3	8	17	match	9	0
.3	0	8	matches	14	4
.3	10	10	matches	9	4
.3	2	16	match	14	0
.3	18	18	match	3	0
.35	2	16	match	9	0
.35	5	5	matches	8	9
.35	10	10	match	5	0
.35	0	3	matches	6	2
.35	20	20	no match	1	0
.35	10	20	match	8	0
.35	12	18	match	8	0
.4	0	20	match	8	1
.4	12	18	match	8	0
.4	5	5	matches	5	2
.4	10	10	match	8	0
.4	20	20	no match	1	0
.4	8	17	match	5	0
Image 2					
.15	0	13	match	4	1
.15	8	3	matches	19	4
.15	0	0	matches	11	6
.15	20	20	match	14	0
.15	0	2	matches	6	3
.15	20	0	matches	33	19
.175	5	4	matches	6	3
.175	11	20	match	3	0
.175	0	20	match	2	0
.175	20	20	match	4	0
.175	3	8	matches	8	3
.175	10	8	matches	6	3
.175	13	13	no match	0	0
.175	0	13	match	2	0
.2	5	5	match	2	0
.2	0	20	no match	0	0
.2	8	3	match	2	0
.2	13	0	match	3	0

With function **select_threshold**:

Threshold	*subimage-x*	*subimage-y*	Result	Number of 5-edge blocks in subimage	Number of 6-edge blocks in subimage
Image 1					
.25	19	0	matches	7	5
.3	20	20	matches	15	2
.3	14	1	matches	10	2
.35	5	5	matches	8	9
.35	2	16	match	9	0
.35	10	10	matches	9	4
.35	0	3	matches	11	7
.4	0	20	match	8	1
.4	10	20	match	8	0
.4	12	18	match	8	0
Image 2					
.1	13	13	matches	20	18
.15	5	5	matches	15	14
.15	0	13	match	4	1
.15	19	19	match	14	0
.15	19	8	matches	14	6
.15	11	20	match	14	0
.15	0	0	matches	12	6
.15	20	11	matches	8	2
.2	8	3	match	2	0
.2	13	0	match	3	0

APPENDIX C

SAMPLE RUNS

```
5 and 5 are the x and y coordinates of the original image point
that is the upper left-hand corner of the subimage.

running calculate_gradient on the orig-image
running calculate_gradient on the subimage
running select_threshold
running calcualte_num_edges_and_unique_index_numbers on the original image
running calculate_num_edges_and_unique_index_numbers on the subimage

The threshold for this run is 0.35.

*subimage-five-edge-list*:
((1 2) (4 3) (3 4) (6 4) (11 10) (10 15) (9 16) (10 16))

*subimage-six-edge-list*:
((2 2) (3 2) (4 2) (2 4) (4 4) (11 8) (12 8) (11 9) (12 9))

*orig-image-five-edge-list*:
((13 3) (12 4) (14 4) (6 7) (9 8) (8 9) (11 9) (16 15) (30 16) (1 19) (2 19) (3 19) (15 20)
 (14 21) (15 21) (13 22) (14 22) (26 22) (12 23) (13 23) (0 26) (0 28) (12 33) (13 33) (25 33
 (26 33) (2 35) (2 36) (3 36))

*orig-image-six-edge-list*:
((7 7) (8 7) (9 7) (7 9) (9 9) (16 13) (17 13) (16 14) (17 14) (0 27))

*******************************************************************************


By comparing nine-block cells with 5 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    5
                                    5

*******************************************************************************

By comparing nine-block cells with 6 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    5
                                    5

*******************************************************************************
*******************************************************************************

19 and 0 are the x and y coordinates of the original image point
that is the upper left-hand corner of the subimage.

running calculate_gradient on the orig-image
running calculate_gradient on the subimage
running select_threshold
running calcualte_num_edges_and_unique_index_numbers on the original image
running calculate_num_edges_and_unique_index_numbers on the subimage

The threshold for this run is 0.25000003.

*subimage-five-edge-list*:
((0 11) (7 11) (4 13) (2 15) (3 15) (4 15) (1 16))

*subimage-six-edge-list*:
((3 14) (4 14) (2 16) (10 16) (11 16))
```

```
*orig-image-five-edge-list*:
((10 0) (10 2) (13 2) (10 3) (12 3) (13 3) (14 4) (7 5) (9 5) (8 6) (9 6) (10 6) (0 7) (10 7)
 (0 8) (8 8) (5 9) (11 9) (7 10) (8 10) (6 11) (9 11) (19 11) (26 11) (8 12) (9 12) (18 12)
 (9 13) (10 13) (13 13) (23 13) (10 14) (18 14) (21 15) (22 15) (23 15) (20 16) (0 17) (13 17)
 (14 17) (17 17) (18 17) (19 17) (21 17) (23 17) (36 17) (0 18) (1 18) (6 18) (7 18) (12 18)
 (13 18) (17 18) (18 18) (24 18) (34 18) (36 18) (3 19) (6 19) (7 19) (32 19) (15 20) (17 20)
 (26 20) (2 21) (15 21) (12 22) (26 22) (27 22) (32 22) (36 22) (12 23) (13 23) (25 23) (0 24)
 (11 24) (12 24) (25 24) (0 25) (4 25) (5 25) (7 25) (4 26) (5 26) (6 26) (13 27) (25 27)
 (26 27) (3 28) (26 28) (34 29) (0 30) (34 30) (36 30) (12 31) (13 31) (32 31) (24 32) (25 32)
 (26 32) (32 32) (36 32) (22 33) (1 34) (11 34) (21 34) (22 34) (23 34) (24 34) (28 34) (1 35)
 (12 35) (35 35) (26 36) (27 36))

*orig-image-six-edge-list*:
((1 0) (11 0) (11 1) (12 2) (10 4) (6 5) (10 5) (11 5) (6 7) (7 7) (9 8) (10 9) (9 10) (7 11)
 (8 11) (17 11) (18 11) (17 12) (11 13) (12 13) (16 13) (18 13) (13 14) (16 14) (22 14) (23 14)
 (16 15) (17 15) (21 16) (29 16) (30 16) (8 18) (25 18) (1 19) (2 19) (16 19) (17 19) (35 19)
 (36 19) (16 20) (35 20) (36 20) (14 21) (32 21) (33 21) (34 21) (35 21) (36 21) (13 22)
 (14 22) (25 22) (0 26) (7 26) (1 27) (2 27) (12 27) (0 29) (2 29) (1 30) (34 31) (36 31)
 (12 32) (13 32) (13 33) (23 33) (27 33) (3 34) (13 34) (25 35) (26 35) (27 35) (3 36) (25 36))

*******************************************************************************

By comparing nine-block cells with 5 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    19
                                    0

*******************************************************************************

By comparing nine-block cells with 6 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    19
                                    0

*******************************************************************************
*******************************************************************************

2 and 16 are the x and y coordinates of the original image point
that is the upper left-hand corner of the subimage.

running calculate_gradient on the orig-image
running calculate_gradient on the subimage
running select_threshold
running calcualte_num_edges_and_unique_index_numbers on the original image
running calculate_num_edges_and_unique_index_numbers on the subimage

The threshold for this run is 0.35.

*subimage-five-edge-list*:
((0 3) (1 3) (13 4) (12 5) (13 5) (11 6) (12 6) (10 7) (11 7))

*subimage-six-edge-list*:
NIL

*orig-image-five-edge-list*:
((13 3) (12 4) (14 4) (6 7) (9 8) (8 9) (11 9) (16 15) (30 16) (1 19) (2 19) (3 19) (15 20)
 (14 21) (15 21) (13 22) (14 22) (26 22) (12 23) (13 23) (0 26) (0 28) (12 33) (13 33) (25 33)
 (26 33) (2 35) (2 36) (3 36))

*orig-image-six-edge-list*:
((7 7) (8 7) (9 7) (7 9) (9 9) (16 13) (17 13) (16 14) (17 14) (0 27))

*******************************************************************************

By comparing nine-block cells with 5 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    2
                                    16
```

```
*****************************************************************************

There are no six-edge blocks in the subimage.

*****************************************************************************
*****************************************************************************

20 and 20 are the x and y coordinates of the original image point
that is the upper left-hand corner of the subimage.

running calculate_gradient on the orig-image
running calculate_gradient on the subimage
running select_threshold
running calcualte_num_edges_and_unique_index_numbers on the original image
running calculate_num_edges_and_unique_index_numbers on the subimage

The threshold for this run is 0.3.

*subimage-five-edge-list*:
((6 0) (13 0) (12 1) (16 1) (5 2) (6 2) (7 2) (6 8) (14 11) (16 11) (12 12) (16 12) (5 13)
 (15 15) (5 16))

*subimage-six-edge-list*:
((12 0) (6 13))

*orig-image-five-edge-list*:
((1 0) (11 0) (11 1) (12 2) (13 2) (13 3) (11 4) (12 4) (14 4) (6 5) (9 6) (10 6) (6 7) (9 8)
 (10 9) (11 9) (9 10) (8 11) (26 11) (9 13) (18 13) (18 14) (23 14) (16 15) (17 15) (29 16)
 (30 16) (13 17) (14 17) (12 18) (13 18) (3 19) (33 19) (34 19) (35 19) (15 20) (26 20) (33 20)
 (2 21) (15 21) (32 21) (36 21) (12 22) (25 22) (26 22) (27 22) (12 23) (13 23) (11 24) (12 24)
 (7 26) (2 28) (26 28) (0 29) (1 29) (34 31) (36 31) (32 32) (36 32) (12 33) (13 33) (25 33)
 (2 34) (35 35) (3 36) (25 36))

*orig-image-six-edge-list*:
((11 2) (7 7) (8 7) (9 7) (7 9) (8 9) (16 13) (16 14) (1 19) (2 19) (32 20) (14 21) (13 22)
 (14 22) (0 26) (1 27) (1 28) (26 33) (2 35) (2 36))

*****************************************************************************


By comparing nine-block cells with 5 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    20
                                    20

*****************************************************************************

By comparing nine-block cells with 6 edges, the program has correctly found
that the upper left-hand corner of the subimage corresponds to the following
x and y coordinates, respectively, of the original image:

                                    20
                                    20

*****************************************************************************
*****************************************************************************
```

APPENDIX D

USER'S MANUAL

A. USING THE PROGRAM

To use this program, you must have an image file that meets the criteria described in Appendix A, and you must have a Symbolics computer with a LISP compiler. Follow these steps:

1. Load the program.
2. Type the command, (**make_image_array "<filename>"**), replacing <filename> with the name of the image file you want to process.
3. Type the command, (**initialize**).
4. If you want the output to go to a file, type (**open_output_file "<filename>"**), again replacing <filename> as appropriate. NOTE: only one such file may be open during any run.
5. Type the command, (**set_up x y**), replacing x and y with the coordinates within the original image of the subimage you wish to process.
6. Type the command, (**process_images**).
7. If you have opened an output file and do not wish to send the results of any other run to that file, type (**close_output_file *fd***). NOTE: you may send as many runs of the program to an output file as you like; if

the file is open during subsequent runs, the results of those runs will be sent to it. Also, if you fail call this function when you are done, you will not be able to access your file.

B. DISPLAYING AN IMAGE

To display an image, use function **display_image** after you have called function **set_up** (Step 5 above). To display the original image, simply type:

(display_image *orig-image-array* 0 0 *size*)

To display the subimage, type:

(display_image *orig-image-array* *subimage-x* *subimage-y* *subimage-size*)

LIST OF REFERENCES

1. Charniak, E. and McDermott, D., Introduction to Artificial Intelligence, pp.89, 99-100, Addison-Wesley Publishing Company, 1985.

2. Sando, J. M., "A Texture Analysis Approach To Computer Vision for Identification of Roads in Aerial Photographs," Master's Thesis, Naval Postgraduate School, Monterey, California, December 1987.

3. Roman, G., Laine, A. F., and Cox, K. C., "Interactive Complexity Control and High-Speed Stereo Matching," Proceedings of the Computer Society Conference on Computer Vision and Pattern Recognition, pp. 171-172, Computer Society Press, 1988.

4. Hartley, C. A., "A Computer Simulation Study of Station Keeping By an Autonomous Submersible Using Bottom-Tracking Sonar," Master's Thesis, Naval Postgraduate School, Monterey, California, June 1988.

5. Encyclopedia Americana, v. 1, pp.217,219, Grolier Incorporated, 1987.

6. Newhall, Beamont, The Airborne Camera, p.63, Hastings House, Publishers, Inc., 1969.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940-5000	1
4.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 98940-5000	2
5.	Professor Neil C. Rowe, Code 52RP Computer Science Department Naval Postgraduate School Monterey, California 93940-5000	3
6.	LCDR J. D. Wolfe Executive Officer NARDAC San Francisco NAS Alameda, California 94501	2

Thesis
W6935 Wolfe
c.1 Determining the location of an observer with respect to aerial photographs.